嵌入式技术与应用丛书

ARM Cortex-M4 + Wi-Fi MCU 应用指南

——CC3200 CCS 基础篇

郭书军　编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书在 CCS 集成开发环境下，以 TI CC3200 LaunchPad 为硬件平台，以 CC3200 SDK v1.1.0 示例程序为核心，在分析示例程序的基础上，介绍 CC3200 各部分的使用方法，并对部分示例程序进行修改和改进，增强程序功能。

全书共 9 章：第 1 章简单介绍 CC3200 的组成，第 2 章介绍 CC3200 的软硬件开发环境，第 3 章介绍 GPIO、UART、SPI 和 I2C 等片内外设接口的应用，第 4 章介绍 GPT、ADC、NVIC 和 μDMA 等片内外设的应用，第 5 章介绍网络应用入门 WLAN 站点和 WLAN 接入点，第 6 章介绍网络应用基础 TCP 和 UDP，第 7 章介绍网络应用配置 SmartConfig 和 WPS，第 8 章介绍 HTTP 服务器，第 9 章介绍睡眠、深睡眠和休眠电源管理。

书后附有 8 个附录，其中包含 CC3200 常用的软/硬件资料，还包含 15 个与书中内容密切相关的实验指导，以方便实验教学。

本书内容新颖，可操作性强，特别适合初学者学习参考，也可以作为嵌入式系统设计和物联网教材，供电子、通信、自动化和物联网等相关专业的师生使用。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

ARM Cortex-M4 + Wi-Fi MCU 应用指南. CC3200 CCS 基础篇 / 郭书军编著．—北京：电子工业出版社，2015.11
（嵌入式技术与应用丛书）

ISBN 978-7-121-27484-8

I．①A…　II．①郭…　III．①微处理器－系统设计　IV．①TP332

中国版本图书馆 CIP 数据核字（2015）第 258104 号

责任编辑：田宏峰
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编　100036
开　　本：787×1 092　1/16　印张：20　字数：500 千字
版　　次：2015 年 11 月第 1 版
印　　次：2015 年 11 月第 1 次印刷
印　　数：3000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

2014 年 6 月，德州仪器（TI）发布了业界第一款内置 Wi-Fi 的 MCU——CC3200 的试制器件 XCC3200HZ 和 SDK v0.5.1，2014 年 9 月 CC3200 的正式器件 CC3200R1 和 SDK v1.0.0 发布，2015 年 3 月 SDK v1.1.0 发布。

为了使用户能够尽快掌握 CC3200 应用，编者从 2014 年 8 月初开始研究 CC3200，并在 IAR 集成开发环境下，以 TI CC3200 LaunchPad 为硬件平台，在 SDK v0.5.1 示例程序的基础上着手编写本书，SDK v1.0.0 发布后，又对已编写内容进行了更新，经过 8 个月的努力，完成了本书初稿。后应 TI 大学计划部的要求，将开发环境更换为 CCS，并趁此机会将 SDK 升级为 v1.1.0，同时对 HTTP 服务器进行了更新和完善。

全书共 9 章，分为 4 个部分：第 1 部分是基础部分，包括第 1 章 CC3200 简介和第 2 章开发环境；第 2 部分是应用 MCU（ARM Cortex-M4）部分，包括第 3 章片内外设接口应用和第 4 章片内设备应用；第 3 部分是 Wi-Fi 网络处理器（CC3100）部分，包括第 5 章到第 8 章；第 4 部分是电源管理部分，包括第 9 章电源管理。

第 1 章：CC3200 简介，简单介绍 CC3200 应用 MCU（ARM Cortex-M4）、Wi-Fi 网络处理器（CC3100）和电源管理 3 大组成部分的特性及引脚复用功能。

第 2 章：开发环境，介绍 CC3200 的软/硬件开发环境。软件开发环境主要包括软件开发工具包（SDK）和 CCS 集成开发环境（IDE），硬件开发环境主要是 TI CC3200 LaunchPad，其中的开包即用体验展示了 CC3200 的特色功能。

第 3 章：片内外设接口应用，介绍 GPIO、UART、SPI 和 I2C 等片内外设接口的应用。以示例程序为核心，在分析示例程序的基础上，介绍 CC3200 各部分的使用方法，并对示例程序进行修改和改进，增强程序功能。

第 4 章：片内设备应用，介绍 GPT、ADC、NVIC 和μDMA 等片内设备的应用。

第 5 章：网络应用入门，介绍网络应用的两个入门示例程序——Wlan_station 示例程序和 Wlan_ap 示例程序，其中 Wlan_station 示例程序可以作为编写网络应用新程序的基础。

第 6 章：网络应用基础，介绍网络应用的两个基础示例程序——Tcp_socket 示例程序和 Udp_socket 示例程序，并在 Wlan_ap 示例程序和 Wlan_station 示例程序的基础上，结合 Tcp_socket 示例程序，进行无线 UART 程序设计。

第 7 章：网络应用配置：介绍网络应用配置 SmartConfig 和 WPS，其中 SmartConfig 是 TI 独特的一步式 Wi-Fi 智能配置技术，允许多个 CC3200 快速高效地连接到 Wi-Fi 网络。

第 8 章：HTTP 服务器：在简单介绍 HTTP 服务器的基础上，介绍 Provisioning_ap 示例程序和 Out_of_box 开包即用示例程序，特别是通过对开包即用示例程序的分析，加深对开包即用体验的理解和掌握。

第 9 章：电源管理：介绍电源管理支持的睡眠、深睡眠和休眠示例程序。

书后附有 8 个附录，其中包含 CC3200 常用的软/硬件资料，还包含与书中内容密切相关的 15 个实验指导，以方便学习和实验教学。

书中所有程序都在 TI CC3200 LaunchPad 和 SDK v1.1.0 环境下调试通过。

在本书的编写过程中，得到德州仪器半导体技术（上海）有限公司潘亚涛先生的大力支持，同时还得到德州仪器的资助，编者在此表示衷心的感谢。

由于内容涉及面广，再加上编者水平有限，书中难免会有错误和不妥之处，敬请广大读者批评指正。编者联系方式（E-mail）：cortex_m3@126.com；QQ 群：CC3200 教材（467209850）。

编　者

2015 年 10 月

目　录

第 1 章　CC3200 简介 …… 1
1.1　应用 MCU …… 2
1.1.1　ARM Cortex-M4 …… 2
1.1.2　存储器 …… 3
1.1.3　片内外设 …… 5
1.2　Wi-Fi 网络处理器（CC3100） …… 6
1.3　电源管理 …… 7
1.4　引脚复用 …… 8
第 2 章　开发环境 …… 9
2.1　软件开发环境 …… 9
2.1.1　软件开发工具包（SDK） …… 9
2.1.2　集成开发环境（IDE） …… 14
2.1.3　实用程序 …… 21
2.1.4　支持工具 …… 24
2.1.5　ROM 服务 …… 30
2.2　硬件开发环境 …… 32
2.2.1　TI CC3200 LaunchPad 开包即用体验 …… 32
2.2.2　服务包更新 …… 35
第 3 章　片内外设接口应用 …… 39
3.1　GPIO 接口应用 …… 39
3.1.1　GPIO 示例程序分析 …… 40
3.1.2　GPIO 使用简介 …… 50
3.1.3　按键控制 LED 闪烁时间程序设计 …… 51
3.1.4　PinMux 使用介绍 …… 52
3.2　UART 接口应用 …… 54
3.2.1　UART 简介 …… 55
3.2.2　UART 示例程序分析 …… 56
3.2.3　UART 使用简介 …… 65
3.2.4　UART 示例程序小改进 …… 66
3.3　SPI 接口应用 …… 66
3.3.1　SPI 示例程序分析 …… 67
3.3.2　SPI 使用简介 …… 78
3.4　I2C 接口应用 …… 79
3.4.1　I2C 示例程序分析 …… 80

3.4.2 I2C 使用简介……89
3.4.3 温度测量和加速度测量程序设计……90
第 4 章 片内外设应用……93
4.1 GPT 应用……93
4.1.1 定时器示例程序分析……94
4.1.2 PWM 示例程序分析……104
4.1.3 计数捕捉示例程序分析……109
4.1.4 GPT 使用简介……115
4.1.5 矩形波周期和脉宽测量程序设计……118
4.2 ADC 应用……120
4.2.1 ADC 示例程序分析……121
4.2.2 ADC 使用简介……126
4.3 NVIC 应用……126
4.3.1 NVIC 示例程序分析……127
4.3.2 NVIC 使用简介……132
4.3.3 外设中断使用小结……133
4.4 μDMA 应用……137
4.4.1 UART DMA 示例程序分析……138
4.4.2 μDMA 使用简介……144
4.4.3 μDMA 示例程序分析……146
第 5 章 网络应用入门……155
5.1 Wlan_station 示例程序分析……156
5.1.1 主函数 main()分析……158
5.1.2 WlanStationMode()分析……160
5.2 Wlan_ap 示例程序分析……173
5.2.1 主函数 main()分析……174
5.2.2 WlanAPMode()分析……175
第 6 章 网络应用基础……183
6.1 Tcp_socket 示例程序分析……184
6.1.1 BsdTcpServer()分析……188
6.1.2 BsdTcpClient()分析……192
6.2 Udp_socket 示例程序分析……195
6.2.1 BsdUdpServer()分析……199
6.2.2 BsdUdpClient()分析……201
6.3 无线 UART 程序设计……204
6.3.1 Wuart_station 程序设计……205
6.3.2 Wuart_ap 程序设计……210
6.3.3 将应用程序编程到串行闪存……214
第 7 章 网络应用配置……217
7.1 Provisioning_smartconfig 示例程序分析……217

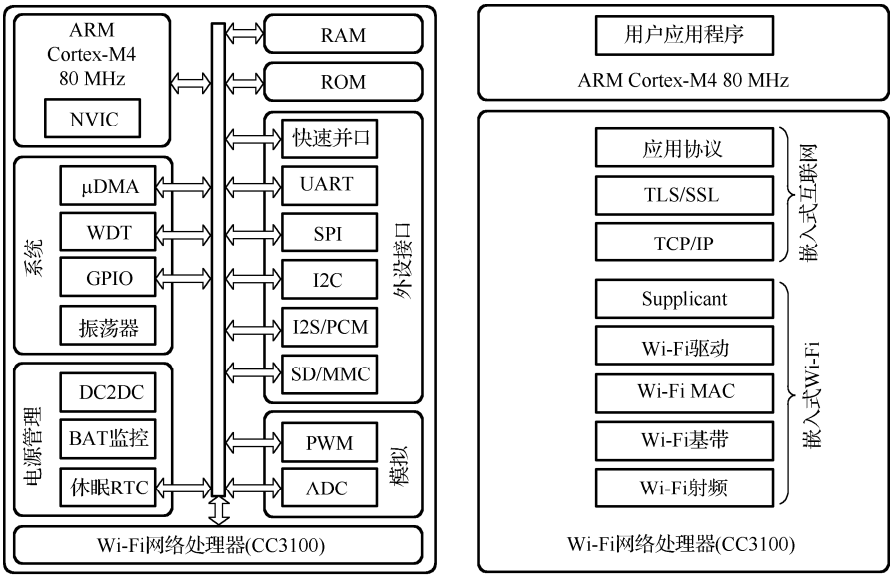
7.2 Provisioning_wps 示例程序分析……221
7.2.1 WpsConnectPushButton()分析……223
7.2.2 WpsConnectPinCode()分析……224
第 8 章 HTTP 服务器……225
8.1 HTTP 服务器简介……225
8.1.1 HTTP GET 处理……226
8.1.2 HTTP POST 处理……227
8.1.3 内部网页……228
8.2 Provisioning_ap 示例程序分析……228
8.2.1 ProvisioningAP()分析……231
8.2.2 GetScanResult()分析……234
8.2.3 HTTP 服务器分析……235
8.3 Out_of_box 示例程序分析……239
8.3.1 HTTP 服务器回调函数分析……242
8.3.2 家庭自动化（Home Automation）网页分析……244
8.3.3 家用电器（Home Appliances）网页分析……245
8.3.4 安全保障（Safety and Security）网页分析……248
8.3.5 智能能源网页分析……249
8.4 HTTP 服务器小结……250
第 9 章 电源管理……253
9.1 Sleep_deepsleep 示例程序分析……253
9.1.1 WDT 唤醒睡眠 PerformPRCMSleepWDTWakeup()分析……254
9.1.2 GPT 唤醒睡眠 PerformPRCMSleepGPTWakeup()分析……256
9.2 Deepsleep_nw 示例程序分析……259
9.3 Hib 示例程序分析……261
附录 A CC3200 外设驱动 API……265
附录 B CC3200 网络驱动 API……275
附录 C CC3200 引脚复用功能……279
附录 D CC3200 外设引脚分配……283
附录 E CC3200 LaunchPad 简介……287
附录 F CC3200 异常和中断……291
附录 G CC3200 μDMA 通道分配……293
附录 H 实验指导……295
使用软件……308
参考文献……309

第 1 章

CC3200 简介

CC3200 是 TI 无线连接 SimpleLink Wi-Fi 和物联网（IoT）解决方案最新推出的一款单片无线 MCU，是业界第一个具有内置 Wi-Fi 的 MCU，是针对物联网应用、集成高性能 ARM Cortex-M4 的无线 MCU。客户能够使用单个集成电路开发整个应用，借助片上 Wi-Fi、互联网和强大的安全协议，无须 Wi-Fi 经验即可实现快速的开发。CC3200 是一个完整平台解决方案，其中包括软件、示例应用、工具、用户和编程指南、参考设计及 TI E2E 支持社区，采用易于布局的四方扁平无引线（QFN）封装。

CC3200 硬件概况和嵌入式软件概况如图 1.1 所示。



(a) 硬件概况　　(b) 嵌入式软件概况

图 1.1　CC3200 硬件概况和嵌入式软件概况

CC3200 包括应用 MCU、Wi-Fi 网络处理器（CC3100）和电源管理 3 大部分。

应用 MCU 包含一个运行频率为 80 MHz 的行业标准 ARM Cortex-M4 内核，包含用于存放代码和数据的内部 RAM 以及存放启动引导程序和外设驱动程序的 ROM，还包含 GPIO、UART、SPI、I2C、I2S/PCM、SD/MMC、PWM 和 ADC 等多种片内外设。

Wi-Fi 网络处理器（CC3100）包含一个 Wi-Fi 片上互联网和一个可完全免除应用 MCU 处理负担的专用 ARM MCU。Wi-Fi 片上互联网包含 Wi-Fi 射频、基带和具有强大加密引擎的 MAC，可以实现支持 256 位加密的快速安全的互联网连接。Wi-Fi 片上互联网还包括嵌入式

TCP/IP 和 TLS/SSL 协议栈、HTTP 服务和多种应用协议。CC3200 支持站点、接入点和 Wi-Fi 直连三种模式，支持 WPA2 个人和企业安全性以及 WPS2。

电源管理包括支持宽电源电压范围的集成直流-直流转换器，可以启用低功耗模式（包括睡眠、深睡眠、低功耗深睡眠和休眠等），具有 RTC 的休眠模式所需电流小于 4 μA。



1.1 应用 MCU

CC3200 应用 MCU 的特性如下。

- ARM Cortex-M4 内核，运行频率 80 MHz。
- 内部存储器。
 - ✧ RAM：存放代码和数据，容量多达 256 KB。
 - ✧ ROM：存放启动引导程序和外设驱动程序，容量为 64 KB。
- 针对高级快速安全性的硬件加密引擎。
 - ✧ AES、DES 和 3DES。
 - ✧ SHA2 和 MD5。
 - ✧ CRC 和校验和。
- 多达 27 个独立可编程、可复用的通用输入输出（GPIO）引脚。
- 2 个通用异步收发器接口（UART）。
- 1 个串行外设接口（SPI）。
- 1 个内部集成电路总线接口（I2C）。
- 1 个多通道音频串行接口（McASP），支持 2 个 I2S 通道。
- 1 个 SD/MMC 接口。
- 8 位并行摄像头接口。
- 4 个通用定时器（GPT），支持 16 位脉冲宽度调制（PWM）模式。
- 4 通道 12 位模/数转换器（ADC）。
- 32 通道微型直接存储器存取（μDMA）。

1.1.1 ARM Cortex–M4

高性能 ARM Cortex-M4 提供低成本的平台，能够实现最小的内存需求，减少引脚数并降低功耗，同时提供卓越的计算性能和系统中断响应。

ARM Cortex-M4 的特性如下。

- 卓越的性能。
 - ✧ 针对嵌入式应用优化的 Thumb 指令集。
 - ✧ 处理程序和线程模式。
 - ✧ 进入和退出中断处理程序时自动保存和恢复处理器状态。
 - ✧ 支持 ARMv6 非对齐访问。
- 嵌套向量中断控制器（NVIC）与处理器内核的紧密结合实现低延迟中断处理。
 - ✧ 3～8 位优先级配置。

- ✧ 动态重新分配中断优先级。
- ✧ 中断优先级分组允许选择中断优先级分组和中断子优先级的数量。
- ✧ 支持咬尾和迟到中断，允许没有状态保存和恢复开销的连续中断处理。
- ✧ 没有指令开销的处理器状态自动保存和恢复。
- ✧ 唤醒中断控制器（WIC）提供超低功耗睡眠模式支持。

● 总线接口。
- ✧ 三个先进高性能总线（AHB）接口：ICode、DCode 和系统总线接口。
- ✧ 存储器和外设位带（Bit-band）支持，包括原子位带读写操作。

● 低成本调试方案。
- ✧ 调试可以访问系统中的所有存储器和寄存器。
- ✧ 支持串行调试端口（SW-DP）和串行 JTAG 调试端口（SWJ-DP）。
- ✧ 闪存补丁和断点（FPB）单元实现代码补丁和断点。

1.1.2　存储器

CC3200 应用 MCU 的存储器包括外部存储器和内部存储器。

1．外部存储器

CC3200 在外部存储器（串行闪存 SFLASH）中保存特有的文件系统，包括服务包文件、系统文件、配置文件、证书文件、网页文件和用户文件等。

用户可以使用格式化命令 API 分配文件系统的总容量，文件系统的起始地址固定在 SFLASH 的开始。应用 MCU 不能直接访问 SFLASH，必须通过文件系统访问分配给文件系统的 SFLASH 区域。

文件系统按照下载顺序管理存储文件 SFLASH 块的分配，这意味着系统中特定文件的位置不固定。存储在 SFLASH 中的文件使用直观的文件名而不是文件标识。文件系统 API 使用纯文本，文件加密和解密用户不可见，加密文件只能通过文件系统进行访问。

文件系统中所有文件类型可以支持多达 128 个文件，文件存储的最小单元是 4 KB，带有安全保障和安全选项的加密文件存储的最小单元是 8 KB，最大的文件大小是 16 MB。

2．内部存储器

内部存储器包括 RAM（静态存储器 SRAM）和 ROM。

（1）SRAM。CC3200 包含片上 SRAM 供应用程序下载和执行，应用程序开发者必须共享 SRAM 用于代码和数据。微型直接存储器存取（μDMA）控制器可以在 SRAM 和外设间传输数据。CC3200 ROM 中拥有丰富的外设驱动程序，可以节省 SRAM 空间。

CC3200 提供多达 256 KB 零等待状态内部 SRAM，能够在低功耗深睡眠（LPDS）模式下有选择地保留，SRAM 在存储器映像中的偏移地址是 0x2000 0000。

（2）ROM。CC3200 的内部零等待状态 ROM 的起始地址是 0x0000 0000，编程有下列组件。

● 启动引导程序（Bootloader）。

- 外设驱动程序（DriverLib）。

启动引导程序用于串行闪存为空时的初始引导程序，外设驱动程序为片内外设提供启动引导能力，实现片内外设的初始化和控制功能，支持查询或中断操作。ROM 中的外设驱动 API 可以被应用程序调用，以减少闪存的需求，多出的闪存可以用于其他目的。

（3）存储器映像。CC3200 存储器映像如表 1.1 所示，其中包含各种外设在存储器的映像地址。

表 1.1 存储器映像

起始地址	结束地址	说　明	注　释
0x0000 0000	0x0000 FFFF	片内 ROM（Bootloader + DriverLib）	64 KB
0x2000 0000	0x2003 FFFF	位带片内 SRAM	256 KB
0x4000 4000	0x4000 4FFF	通用输入输出接口 GPIOA0	见 3.1 节
0x4000 5000	0x4000 5FFF	通用输入输出接口 GPIOA1	见 3.1 节
0x4000 6000	0x4000 6FFF	通用输入输出接口 GPIOA2	见 3.1 节
0x4000 7000	0x4000 7FFF	通用输入输出接口 GPIOA3	见 3.1 节
0x4000 C000	0x4000 CFFF	串行异步收发器接口 UARTA0	见 3.2 节
0x4000 D000	0x4000 DFFF	串行异步收发器接口 UARTA1	见 3.2 节
0x4002 0000	0x4002 07FF	内部集成电路总线接口 I2CA0（主模式）	见 3.4 节
0x4002 0800	0x4002 0FFF	内部集成电路总线接口 I2CA0（从模式）	见 3.4 节
0x4003 0000	0x4003 0FFF	通用定时器 GPTA0	见 4.1 节
0x4003 1000	0x4003 1FFF	通用定时器 GPTA1	见 4.1 节
0x4003 2000	0x4003 2FFF	通用定时器 GPTA2	见 4.1 节
0x4003 3000	0x4003 3FFF	通用定时器 GPTA3	见 4.1 节
0x400F E000	0x400F EFFF	系统控制	
0x400F F000	0x400F FFFF	微型直接存储器存取（μDMA）	见 4.4 节
0x4401 C000	0x4401 EFFF	多通道音频串行接口（McASP）	
0x4402 0000	0x4402 0FFF	串行闪存串行外设接口 SSPI	外部串行闪存使用
0x4402 1000	0x4402 2FFF	通用串行外设接口 GSPI	应用 MCU 使用见 3.3 节
0x4402 5000	0x4402 5FFF	MCU 复位时钟管理	
0x4402 6000	0x4402 6FFF	MCU 配置空间	
0x4402 D000	0x4402 DFFF	全局电源、复位和时钟管理（GPRCM）	
0x4402 E000	0x4402 EFFF	MCU 共享配置	
0x4402 F000	0x4402 FFFF	休眠配置	
0x4403 0000	0x4403 FFFF	加密范围（包括下列 4 个加密相关模块）	
0x4403 0000	0x4403 0FFF	DTHE 寄存器和 TCP 校验和	
0x4403 5000	0x4403 5FFF	安全散列算法 SHA/消息摘要算法 MD5	
0x4403 7000	0x4403 7FFF	高级加密标准 AES	
0x4403 9000	0x4403 9FFF	数据加密标准 DES	
0xE000 0000	0xE000 0FFF	测量跟踪宏单元（ITM）	
0xE000 1000	0xE000 1FFF	数据观察点和跟踪（DWT）	
0xE000 2000	0xE000 2FFF	闪存补丁和断点（FPB）	
0xE000 E000	0xE000 EFFF	嵌套向量中断控制器（NVIC）	见 4.3 节
0xE004 0000	0xE004 0FFF	跟踪端口接口单元（TPIU）	

1.1.3　片内外设

CC3200 支持下列片内外设接口和外设。

- 通用输入输出接口（GPIO）。
- 串行异步收发器接口（UART）。
- 串行外设接口（SPI）。
- 内部集成电路总线接口（I2C）。
- 多通道音频串行接口（McASP）。
- SD/MMC 接口。
- 并行摄像头接口。
- 通用定时器（GPT）。
- 模/数转换器（ADC）。
- 微型直接存储器存取（μDMA）。

（1）通用输入输出接口（GPIO）。通用输入输出接口（GPIO）分为 4 组，每组 8 个。根据外设用途的不同，GPIO 支持 27 个可编程 GPIO 引脚，每个 GPIO 引脚可配置为 10 μA 上拉或下拉，驱动能力可配置为 2 mA、4 mA 或 6 mA，也可配置为开漏输出。

（2）串行异步收发器接口（UART）。串行异步收发器接口（UART）具有可编程波特率发生器，允许速率高达 3 Mbps，标准的异步通信起始位、停止位和奇偶校验位，使用μDMA 可以实现高效传输。

（3）串行外设接口（SPI）。串行外设接口（SPI）可以配置为主设备或从设备，主设备串行时钟的频率、极性和相位可编程，片选和外部时钟的定时控制可编程，第一个发送字前的延时可编程。

（4）内部集成电路总线接口（I2C）。内部集成电路总线接口（I2C）通过两根串行线可以在单个接口上连接多种 I2C 外设，也可以工作在标准模式（100 kbps）或者快速模式（400 kbps）。

（5）多通道音频串行接口（McASP）。多通道音频串行接口（McASP）作为通用音频串行接口，为多通道音频应用优化，支持通过两个数据引脚进行立体声传输，发送和接收部分可同步工作。

（6）SD/MMC 接口。SD/MMC 接口用于连接大容量 SD 卡存储器，支持 1 位数据模式。

（7）并行摄像头接口。并行摄像头接口可以连接各种外部图像传感器，图像数据存放在 FIFO 中，可以产生 DMA 请求，数据宽度是 8 位。

（8）通用定时器（GPT）。通用定时器（GPT）可以对输入引脚上的外部事件进行计数或计时，并可以产生脉冲宽度调制（PWM）信号从输出引脚上输出。

（9）模/数转换器（ADC）。模/数转换器（ADC）的主要功能是将连续的模拟信号转化为离散的数字信号，以便于微控制器进行数据处理。

（10）微型直接存储器存取（μDMA）。微型直接存储器存取（μDMA）将 Cortex-M4 从数据传输任务中解脱出来，实现外设和存储器之间或者存储器和存储器之间的批量数据传输。

1.2 Wi-Fi 网络处理器（CC3100）

CC3200 Wi-Fi 网络处理器（CC3100）可以提供快速安全的 WLAN 和因特网连接，支持站点、接入点和 Wi-Fi 直连三种模式，特性如下。

- 特有 Wi-Fi 片上互联网（Internet-On-a-Chip）。
- 专用 ARM MCU，完全免除应用 MCU 的 Wi-Fi 和互联网协议处理负担。
- Wi-Fi 和互联网协议存放在 ROM 中。
- 包含 802.11b/g/n 射频、基带、MAC、Wi-Fi 驱动和 Supplicant。
- TCP/IP 协议栈。
 - ✧ 行业标准 BSD 套接字应用编程接口（API）。
 - ✧ 同时 8 个 TCP 或 UDP 套接字。
 - ✧ 同时 2 个 TLS 和 SSL 套接字。
- 强大的加密引擎，可以实现支持 256 位 AES 加密的快速安全的互联网连接。
- 站点（STA）、接入点（AP）和 Wi-Fi 直连（P2P）模式。
- WPA2 个人和企业安全性。
- 用于自主和快速 Wi-Fi 连接的 SimpleLink 连接管理器。
- 用于简单灵活 Wi-Fi 配置的智能配置（SmartConfig）技术、AP 模式和 WPS2。
- 发射功率。
 - ✧ 18.0 dBm @ 1 DSSS（直序扩频）。
 - ✧ 14.5 dBm @ 54 OFDM（正交频分复用）。
- 接收灵敏度。
 - ✧ –95.7 dBm @ 1 DSSS（直序扩频）。
 - ✧ –74.0 dBm @ 54 OFDM（正交频分复用）。

Wi-Fi 网络处理器（CC3100）支持的主要特性如表 1.2 所示。

表 1.2 Wi-Fi 网络处理器（CC3100）支持的主要特性

特　性	类　别	域	说　明
IPv4	网络协议栈	TCP/IP	IPv4 协议栈
TCP/UDP	网络协议栈	TCP/IP	基本协议
TLS/SSL	安全	TCP/IP	TLS v1.2（客户端/服务器）/SSL v3.0
DHCP	协议	TCP/IP	客户端和服务器模式
mDNS	应用	TCP/IP	多播域名服务
HTTP	应用	TCP/IP	URL 静态和动态响应模版
Policies	连接	WLAN	允许管理连接和重连接策略
WPS2	配置	WLAN	初始化产品配置 PBC 和 PIN 方法
AP Config	配置	WLAN	初始化产品配置接入点方式（配置网页和信标信息元）
SmartConfig	配置	WLAN	初始化产品配置的替代方法
Station	角色	WLAN	具有传统 802.11 节电功能的 802.11b/g/n 站点
AP	角色	WLAN	具有传统 802.11 节电功能的 802.11b/g 接入点
P2P	角色	WLAN	P2P 客户端

1.3　电 源 管 理

CC3200 电源管理的特性如下。

- 集成直流-直流转换器，支持宽范围的电源电压。
 - ✧ VBAT 宽电压模式：2.1～3.6 V。
 - ✧ 预稳压 1.85 V 模式。
- 高级低功耗模式。
 - ✧ 休眠（Hibernate）：4 μA。
 - ✧ 低功耗深睡眠（LPDS）：120 μA。
 - ✧ RX（MCU 活动）：59 mA @ 54 OFDM。
 - ✧ TX（MCU 活动）：229 mA @ 54 OFDM，最大功率。
 - ✧ 空闲连接（MCU 处于 LPDS）：695 μA @ DTIM = 1。

从电源管理的角度看，CC3200 的应用 MCU 和 Wi-Fi 网络处理器（CC3100）是独立的，应用 MCU 运行从外部串行闪存装入的用户应用程序，Wi-Fi 网络处理器（CC3100）运行预编程的 TCP/IP 和 Wi-Fi 数据链路层函数。

用户程序控制应用 MCU 的电源状态，可以有下列 4 种电源模式。

- MCU 活动（Active）模式：MCU 以 80 MHz 主频执行程序。
- MCU 睡眠（Sleep）模式：MCU 时钟被禁止，设备状态被保留。睡眠模式提供即时唤醒功能，可以通过内部定时器、GPIO 引脚或外设唤醒。
- MCU 低功耗深睡眠（LPDS）模式：状态信息丢失，只有某些 MCU 特定的寄存器配置被保留。MCU 可以通过内部定时器或 GPIO0～6 上的外部事件唤醒，唤醒时间小于 3 ms。存储器的某些部分可以被保留，保留存储器的数量可配置，用户可以选择保存代码或 MCU 特定的设置。
- MCU 休眠（Hibernate）模式：最低功耗模式，只有一小部分直接由输入电源供电的逻辑被保留，实时时钟（RTC）保持运行，MCU 可以通过 RTC 超时或 GPIO0～6 上的外部事件唤醒，唤醒时间比 LPDS 模式长 15 ms 加上从串行闪存加载应用程序的时间。

Wi-Fi 网络处理器（CC3100）可以按照自己的模式活动或低功耗深睡眠，当没有网络活动时，Wi-Fi 网络处理器（CC3100）大部分时间睡眠，只有接收信标时唤醒。

Wi-Fi 网络处理器（CC3100）有以下 6 种电源模式。

- 网络活动模式（处理低 3 层）：发送或接收 IP 协议数据包。
- 网络活动模式（处理低 2 层）：发送或接收 MAC 管理帧，不需要 IP 处理。
- 网络活动监听模式：特殊的功耗优化活动模式，用于接收信标帧，不支持其他帧。
- 网络连接空闲模式：实现 802.11 节能操作的复合模式。网络处理器（CC3100）在信标间自动进入 LPDS 模式，接收信标时进入活动监听模式，通过信标确定接入点是否有数据传送，如果没有，网络处理器（CC3100）返回 LPDS 模式。

- 网络低功耗深睡眠（LPDS）模式：信标间的低功耗状态，状态由网络处理器（CC3100）保留，允许快速唤醒。
- 网络禁用模式：网络处理器（CC3100）不可用。

应用 MCU 和网络处理器（CC3100）的操作应确保设备大部分时间停留在最低功耗模式，以延长电池寿命。CC3200 重要的片级电源模式如表 1.3 所示。

表 1.3 片级电源模式

电源状态	网络活动模式	网络 LPDS 模式	网络禁用模式
MCU 活动模式	芯片 = 活动（C）	芯片 = 活动	芯片 = 活动
MCU LPDS 模式	芯片 = 活动（A）	芯片 = LPDS（B）	芯片 = LPDS
MCU 休眠模式	不支持	不支持	芯片 = 休眠（D）

应用程序中电源模式的使用例子如下。

（1）设备一直连接到网络，但只发送和接收少量数据，大部分时间连接空闲：使用模式 A（接收信标帧）和模式 B（等待下一个信标）。

（2）设备不一直连接到网络，而是周期唤醒（如间隔 10 min）发送数据，大部分时间在模式 D（休眠）：短暂跳到模式 C 传输数据。

1.4 引 脚 复 用

CC3200 广泛使用引脚复用功能，以便在尽可能小的封装内容纳大量的外设。为了实现这种结构，引脚复用使用硬件配置和寄存器控制相结合的方法进行控制。

注意：TI 强烈建议使用 CC3200 引脚复用实用工具 PinMux 获得需要的引脚配置，PinMux 的使用方法参考 3.1.4 节。

引脚复用需要特别注意下列事项。

- 所有 I/O 引脚支持 2 mA、4 mA 或 6 mA 驱动能力，每个引脚的驱动能力独立配置。
- 所有 I/O 引脚支持 10 μA 上拉和下拉，默认不激活。
- 所有 I/O 引脚在设备处于休眠状态时保持浮空。
- VIO 和 VBAT 必须始终接在一起。

CC3200 引脚复用功能参见附录 C，外设引脚分配参见附录 D。

第 2 章

开 发 环 境

CC3200 开发环境包括软件开发环境和硬件开发环境。

2.1 软件开发环境

CC3200 软件开发环境包括软件开发工具包（SDK）、集成开发环境（IDE）、实用程序和支持工具等。

2.1.1 软件开发工具包（SDK）

CC3200 软件开发工具包（SDK）目前有四个版本：用于试制器件的 SDK v0.5.1 和 SDK v0.5.2，以及用于正式器件的 SDK v1.0.0 和 SDK v1.1.0。下面以用于正式器件的 SDK v1.1.0 为例介绍 SDK 的安装和目录结构。

1．SDK 安装

CC3200 SDK v1.1.0 的安装文件是“CC3200SDK-1.1.0-windows-installer.exe”，该安装文件支持 Windows XP、Windows 7 和 Windows 8，下面以 Windows 7 为例介绍 CC3200 SDK v1.1.0 的安装步骤。

（1）双击“CC3200SDK-1.1.0-windows-installer.exe”，显示图 2.1 所示安装向导。



图 2.1　CC3200 SDK v1.1.0 安装向导

（2）单击“Next”（下一步）按钮，显示如图 2.2 所示的许可协议。

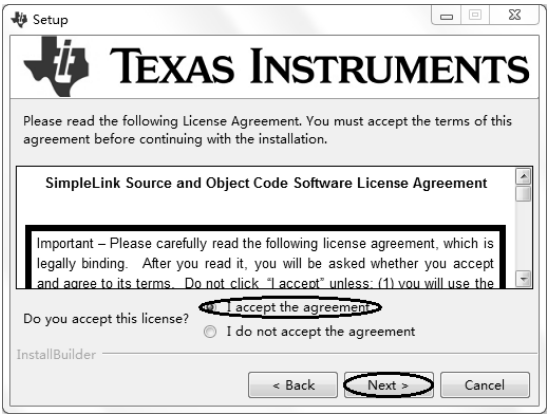


图 2.2 CC3200 SDK v1.1.0 许可协议

（3）阅读许可协议，选中“I accept the agreement”（我接受此协议），单击“Next”（下一步）按钮，显示如图 2.3 所示的安装目录。

（4）接受默认安装目录或修改安装目录，单击“Next”（下一步）按钮两次，开始安装 CC3200 SDK。安装完成后，显示如图 2.4 所示的对话框。



图 2.3 CC3200 SDK v1.1.0 安装目录

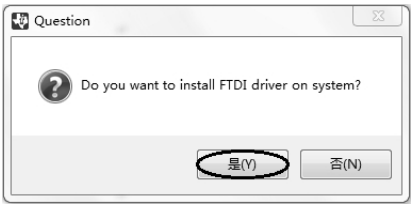


图 2.4 安装 FTDI 驱动程序问题对话框

（5）单击“是（Y）”按钮，安装 FTDI 驱动程序。FTDI 驱动程序安装完成后，显示如图 2.5 所示的界面。

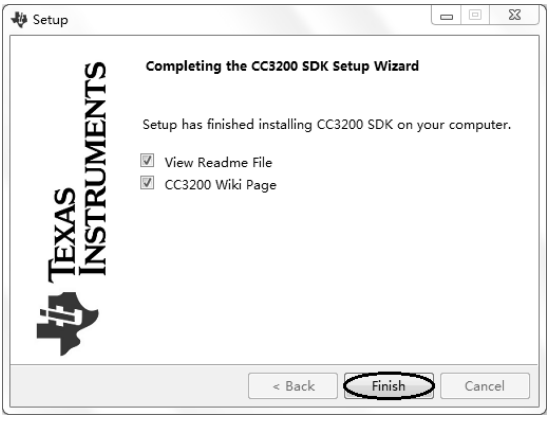


图 2.5 CC3200 SDK v1.1.0 安装完成

（6）单击“Finish”（完成）按钮完成安装，显示 Readme File（自述文件）和 CC3200 Wiki Page（维基网页），维基网页中有 CC3200 的详细介绍。

2．SDK 目录结构

CC3200 SDK 目录结构及其内容说明如表 2.1 所示。

表 2.1　CC3200 SDK 目录结构及其内容说明

目录名	说　　明
docs	驱动程序库指南和示例程序应用笔记等
driverlib	外设驱动程序库源代码
example	示例程序源代码
inc	寄存器定义头文件
middleware	易于使用的电源管理架构解决方案
netapps	HTTP、SMTP、TFTP 和 XMPP 客户端库源文件
oslib	配置 Free-RTOS 和 TI-RTOS 的接口文件
simplelink	SimpleLink 网络处理器（CC3100）驱动程序代码
simplelink_extlib	OTA（Over the Air）库
third_party	FatFS 和 Free-RTOS 源文件
ti_rtos	TI-RTOS 抽象层文件
tools	FTDI 驱动程序以及 CCS 和 IAR 补丁程序等

（1）外设驱动程序库。CC3200 外设驱动程序库简要说明如表 2.2 所示。

表 2.2　外设驱动程序库简要说明

驱动程序库名	说　　明	驱动程序库名	说　　明
adc	模/数转换器 API，ADC 有 4 个输入通道，最大输入电压 1.4 V	gpio	通用输入输出 API，GPIO 分为 4 组，每组 8 个
camera	摄像头快速并行接口 API，8 位并行接口	crc	循环冗余校验 API
des	数据加密标准 API	i2c	内部集成电路接口 API
aes	高级加密标准 API	udma	微型直接存储器存取 API
i2s	内部集成电路音频接口 API	interrupt	中断 API
pin	引脚复用设置 API	prcm	电源、复位和时钟模块 API
sdhost	安全数字主控制器 API	shamd5	安全散列算法、消息摘要算法 API
spi	串行外设接口 API	systick	系统时钟定时器 API
timer	通用定时器 API	uart	通用异步收发器接口 API
wdt	看门狗定时器 API		

CC3200 外设驱动 API 参见附录 A，详细信息和使用方法参见文档“docs\CC3200 Peripheral Driver Library User's Guide.chm”。

（2）网络驱动程序库。TI SimpleLink 架构提供从基本设备管理到无线网络配置和 BSD 套接字服务等一系列功能，为了更好地进行设计，这些功能被分离成单独的模块，每个模块代表不同的 SimpleLink 架构功能。SimpleLink 架构组件如表 2.3 所示。

表 2.3 SimpleLink 架构组件

组 件	功 能
device	初始化主机，控制与网络处理器（CC3100）的通信
wlan	连接到接入点、扫描接入点、添加/删除接入点配置文件、WLAN 安全
socket	TCP/UDP 客户端套接字、服务器套接字、Rx/Tx
netapp	DNS 解析、Ping 远程设备、地址解析协议
netcfg	IP/MAC 地址配置
fs	文件系统读写

CC3200 网络驱动 API 参见附录 B，详细信息和使用方法参见网页“docs\simplelink_api\programmers_guide.html”。

（3）电源模式。MCU 电源管理架构库允许用户应用程序利用下列电源管理模式。

- 睡眠（Sleep）。
- 深睡眠（Deep Sleep）。
- 低功耗深睡眠（LPDS）。
- 休眠（Hibernate）。

Wi-Fi 网络处理器（CC3100）支持下列低功耗策略。

- 正常（默认）：数据传送时间和功耗性能的最佳折中。
- 低功耗：仅用于收发（Transceiver）模式。
- 长睡眠间隔：在配置睡眠间隔（最长 2 s）后的下一个 DTIM 唤醒，仅用于客户端套接字（Client Socket）模式。

（4）示例程序。CC3200 SDK 的 example 目录中包含 62 个示例程序目录，这些示例程序大致分为两类：22 个 MCU 示例程序和 40 个网络示例程序，两类示例程序的目录名及其简要说明分别如表 2.4 和表 2.5 所示。

表 2.4 MCU 示例程序简要说明

目 录 名	说 明
adc	使用 Driverlib API 演示 ADC 模块的功能
aes	演示 AES Driverlib API 的使用
blinky	演示 GPIO Driverlib API 的使用
crc	演示 CRC Driverlib API 的使用
des	演示 DES Driverlib API 的使用
freertos_demo	演示 Free-RTOS 功能：多任务创建和使用队列的任务间通信
i2c_demo	演示 I2C Driverlib API 的使用：读写 I2C 器件
interrupt	演示中断 Driverlib API 的使用：中断优先级和咬尾中断
pwm	演示通用定时器的 PWM 功能
sdhost	演示 SD 卡控制器的初始化和 SD 卡数据块的读写
sdhost_fatfs	演示使用 FatFS 实现 SD 卡的块级读写操作
shamd5	演示 SHA-MD5 Driverlib API 的使用
sleep_deepsleep	演示 MCU 的睡眠和深睡眠功能
spi_demo	演示全双工 4 线主从模式 SPI 的初始化和数据的发送与接收
timer	演示 Timer Driverlib API 的使用：16 位定时器中断切换 GPIO 的状态
timer_cc	演示使用定时器的计数捕捉特性测量外部信号的频率
uart_demo	演示 UART Driverlib API 的使用：终端回显
uart_dma	演示 UART 的 DMA 和中断功能
udma	演示 DMA Driverlib API 的使用：各种 DMA 功能
watchdog	演示 WDT Driverlib API 的使用：系统故障时复位 MCU
application_bootloader	演示管理应用程序映像更新的启动引导操作（v1.1.0）
dynamic_lib_loader	演示程序运行时从非易失存储器动态加载二进制应用程序的方法（v1.1.0）

表 2.5　网络示例程序简要说明

目　录　名	说　　明
antenna_selection	演示使用网页浏览器选择天线，增强接入点的信号强度
camera_application	演示通过设备中的网页浏览器捕捉图像，需要 Camera BoosterPack 配件
connection_policy	演示连接到接入点的策略
deepsleep_nw	演示网络中深睡眠节电方法的使用
email	通过 SMTP 发送预先配置的 E-mail
ent_wlan	演示使用 SFLASH 中的证书连接企业网络
file_download	演示从网页服务器下载文件到设备存储器
file_operations	演示文件系统 API 的使用
get_time	连接到 SNTP 服务器获取时间信息
get_weather	连接到开放的天气服务器获取天气数据
getting_started_wlan_with_ap	WLAN 接入点入门，设备配置为接入点（AP）模式，通过 Ping 客户端验证连接
getting_started_wlan_with_station	WLAN 站点入门，设备配置为站点（STA）模式，通过 Ping 网关和外部主机验证连接
hib	演示网络中休眠节电方法的使用
httpserver	演示 HTTP 服务
idle_profile	使用电源管理架构中间件实现低功耗深睡眠（LPDS）
mdns	演示 MDNS 功能的使用
mode_config	演示站点（STA）和接入点（AP）模式的切换
nwp_filters	允许用户定义和管理接收过滤方法，以减少传送到主机的数据量，降低功耗
out_of_box	通过网页浏览器查看不同的演示和 SDK 网页连接
p2p	演示 Wi-Fi 直连特性
provisioning_ap	演示 AP 配置技术的使用
provisioning_smartconfig	演示 TI SmartConfig Wi-Fi 配置技术的使用
provisioning_wps	演示 WPS Wi-Fi 配置技术的使用
scan_policy	演示扫描策略
sensor_profile	使用电源管理架构中间件实现休眠（Hibernate）
serial_wifi	UART Wi-Fi 转换，通过 UART 接口实现方便独立的无线终端访问
ssl	演示 SSL 证书的使用
tcp_socket	演示基本的 TCP 功能：发送和接收 TCP 数据包
transceiver_mode	演示在 Wi-Fi 物理层上建立专有的协议，灵活的建立自己的数据包
udp_socket	演示基本的 UDP 功能：发送和接收 UDP 数据包
wifi_audio_app	演示双向音频应用，需要 Audio BoosterPack 配件
xmpp_client	演示与 XMPP 服务器的连接
tftp_client	演示从 TFTP 服务器读文件和向 TFTP 服务器写文件（v1.0.0）
watchdog_system_demo	演示系统故障时复位 MCU 和网络处理器（v1.0.0）
websock_camera	演示使用摄像头捕捉 VGA 帧并通过 websock 协议发送到客户端（v1.0.0）
http_client_demo	演示 HTTP 客户端库使用（v1.1.0）
idle_profile_nonos	无操作系统环境下使用电源管理架构中间件实现低功耗深睡眠（v1.1.0）
mqtt_client	演示 MQTT 客户端使用（v1.1.0）
mqtt_server	演示 MQTT 服务器使用（v1.1.0）
ota_update	演示服务包 OTA 更新（v1.1.0）

本书将以这些示例程序为核心，在示例程序分析的基础上，介绍各部分的使用方法，并对部分示例程序进行修改和改进，增强程序功能。

2.1.2 集成开发环境（IDE）

CC3200 的集成开发环境（IDE）可以使用 CCS（v6.1.0）、IAR（v7.30）或 GCC（Linux 环境），下面以 CCS（v6.1.0）为例介绍集成开发环境的安装和使用。

1．CCS 安装

CCS 有在线和离线两种安装方法，CCS v6.1.0 的离线安装压缩文件是“CCS6.1.0.00104_win32.zip”，将压缩文件解压缩后得到“CCS6.1.0.00104_win32”目录，其中包含安装文件“ccs_setup_6.1.0.00104.exe”，安装步骤如下。

（1）双击 ccs_setup_6.1.0.00104.exe，显示如图 2.6 所示的许可协议。



图 2.6　CCS 许可协议

（2）阅读许可协议，选中“I accept the terms of the license agreement”（我接受许可协议中的条款），单击“Next”（下一步）按钮，显示如图 2.7 所示的安装目录。

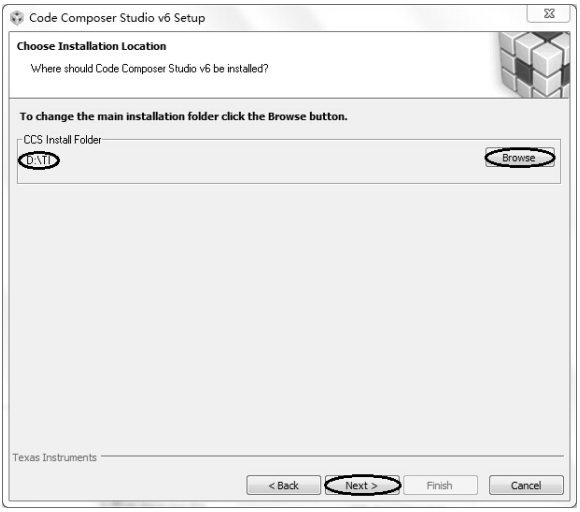


图 2.7　CCS 安装目录

（3）接受默认安装目录或单击“Browse”（浏览）按钮修改安装目录，单击“Next”（下一步）按钮，显示如图 2.8 所示的处理器支持。

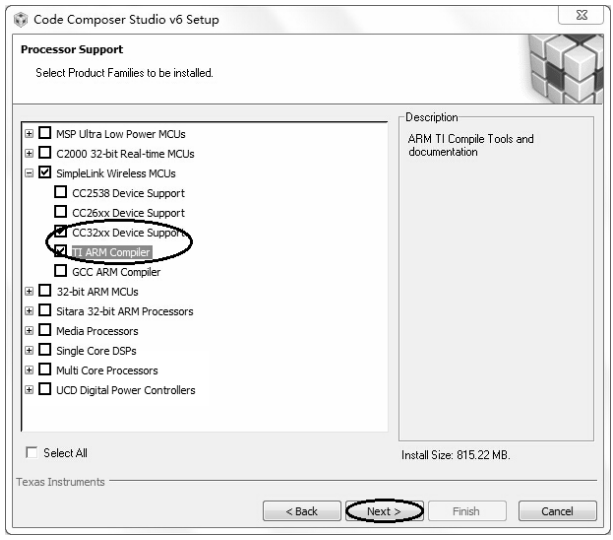


图 2.8　CCS 处理器支持

（4）选择“SimpleLink Wireless MCUs”下的“CC32xx Device Support”和“TI ARM Compiler”，单击“Next”（下一步）按钮，显示如图 2.9 所示的选择调试探头对话框。

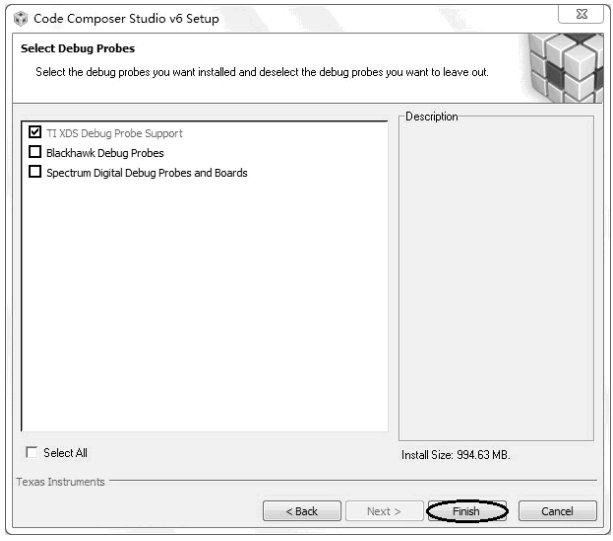


图 2.9　CCS 选择调试探头

（5）接受默认设置，单击“Finish”（完成）按钮开始安装，安装过程大约需要几分钟。安装完成后，显示如图 2.10 所示的安装成功对话框。

（6）单击“Finish”（完成）按钮完成安装，创建快捷方式并执行 CCS，显示如图 2.11 所示的选择工作空间对话框。

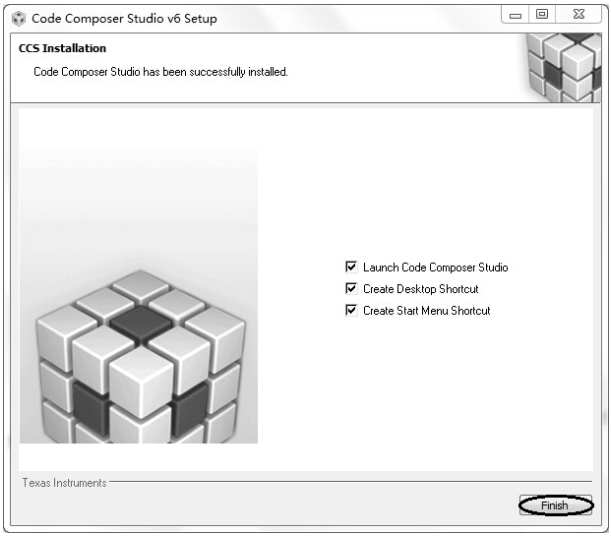


图 2.10　CCS 安装成功

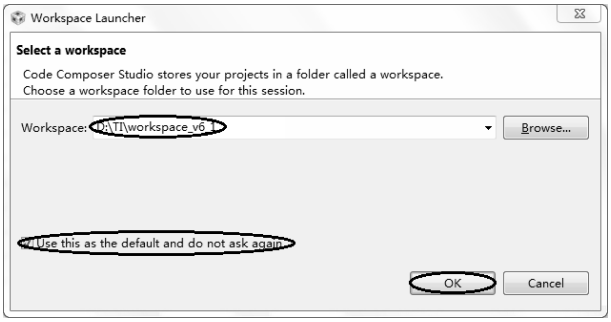


图 2.11　CCS 选择工作空间

（7）接受默认设置或修改工作空间目录，选择“Use this as the default and do not ask again”，单击“OK”按钮，显示如图 2.12 所示的 CCS 编辑界面。



图 2.12　CCS 编辑界面

正常使用时，单击“File”（文件）→“Switch Workspace”（更换工作空间）→“Other…”（其他），可以随时更换工作空间，更换工作空间后 CCS 将重新启动。

（8）单击“Getting Started”（开始）标签右上角的“App Center”（应用中心），打开应用中心标签，在搜索框中输入“cc3200”，在搜索框右边的下拉菜单中选择“All”，搜索出“CC3200 Add-On”和“TI-RTOS for SimpleLink”插件，单击插件右下角的“Select”选中插件进行更新，如图 2.13 所示。

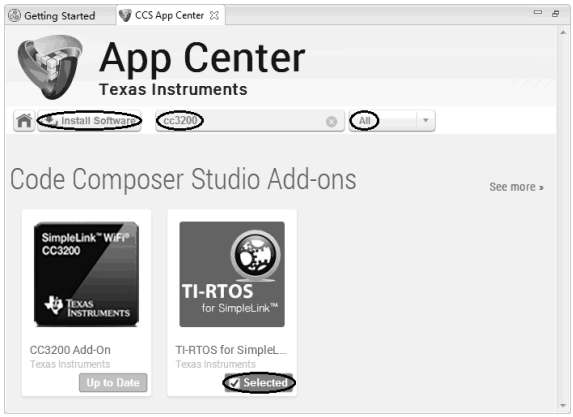


图 2.13　CCS 应用中心

如果插件右下角显示“Up to Date”（最新），则不需要更新。

（9）单击应用中心左上角的“Install Software”（安装软件），安装选择的插件。安装过程中显示许可协议对话框时，单击“I accept the terms of the license agreement”（我接受许可协议中的条款），单击“OK”按钮。

（10）安装完成后，显示“Software Updates”（软件更新）对话框，单击“Yes”（是）按钮重新启动 CCS，如图 2.14 所示。



图 2.14　CCS 软件更新对话框

2．CCS 使用

CCS 使用的主要内容是编辑源文件、生成和调试目标文件等，其中最主要的是调试目标文件。下面以导入项目为例，简单介绍 CCS 的使用。

（1）单击“Project”（项目）→“Import CCS Projects…”（导入 CCS 项目），打开如图 2.15 所示的导入 CCS 项目对话框。

（2）单击“Browse”（浏览）按钮，选择 CC3200 SDK 安装目录，其中的 CCS 项目显示在“Discovered projects”（发现项目）下。

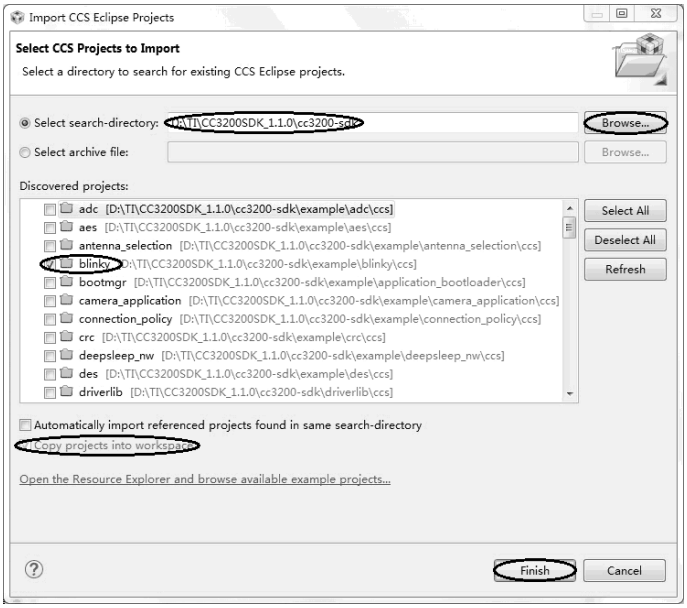


图 2.15　导入 CCS 项目对话框

选择“blinky”项目，选项“Copy projects into workspace”（复制项目到工作空间）自动选中，且变为灰色（不允许修改）。

单击“Finish”按钮，CCS 在工作空间目录中创建 blinky 目录，并将 SDK 中的 blinky 项目文件复制到 blinky 目录，项目出现在 CCS 的“Project Explorer”（项目浏览器）视图中（单击项目名左边的三角符号可以展开项目），如图 2.16 所示。

（3）右击“Project Explorer”（项目浏览器）中的“blinky”项目，在弹出菜单中单击“Properties”（属性），打开“属性”对话框，确认“Device”（器件）选择“CC3200”，选择“Compiler version”（编译器版本）为“TI v5.2.2”，如图 2.17 所示。

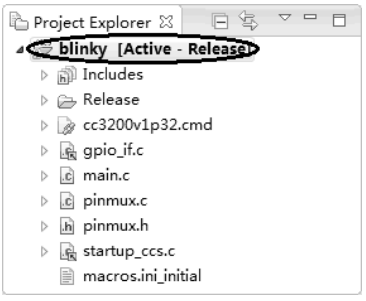


图 2.16　CCS 项目浏览器

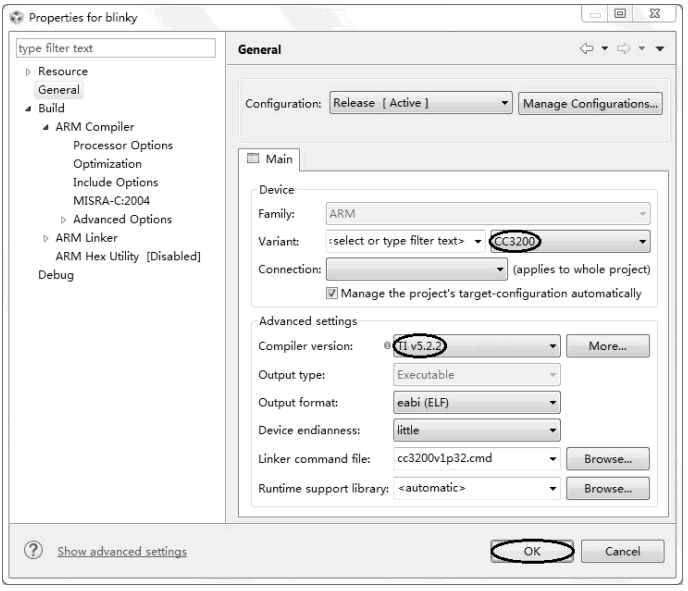


图 2.17　CCS 项目属性对话框

（4）右击“Project Explorer”（项目浏览器）中的“blinky”项目，在弹出菜单中单击“Build Project”（生成项目）生成项目目标文件“blinky.out”。

注意：第一次生成项目时，CCS 需要生成编译库，因此生成时间大约需要几分钟。

为了调试目标文件，必须进行目标配置，步骤如下。

（1）单击“View”（视图）→“Target Configuration”（目标配置），打开如图 2.18 所示的目标配置视图。

（2）右击“User Defined”（用户定义），在弹出菜单中单击“Import Target Configuration”（导入目标配置），选择“CC3200 SDK”安装目录“tools\ccs_patch”目录中的“CC3200.ccxml”文件，在“File Operation”（文件操作）对话框中单击“OK”按钮复制文件，“CC3200.ccxml”出现在“User Defined”（用户定义）下，如图 2.19 所示。

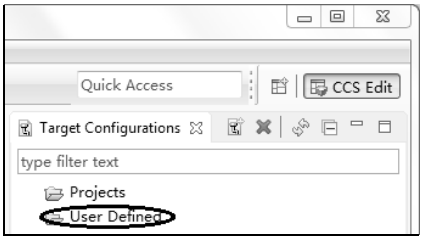


图 2.18　CCS 目标配置 1

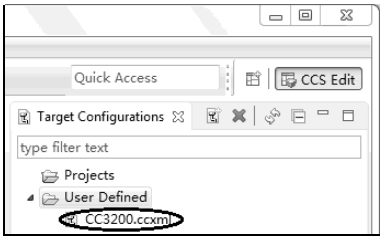


图 2.19　CCS 目标配置 2

（3）右击“CC3200.ccxml”，在弹出菜单中单击“Set as Default”（设为默认值），将“CC3200.ccxml”设为默认配置。

（4）用 USB 线将开发板连接到计算机，单击“Run”（运行）→“Debug”（调试）或按“F11”键，CCS 将目标文件“blinky.out”下载到 CC3200 并进入调试界面，如图 2.20 所示。

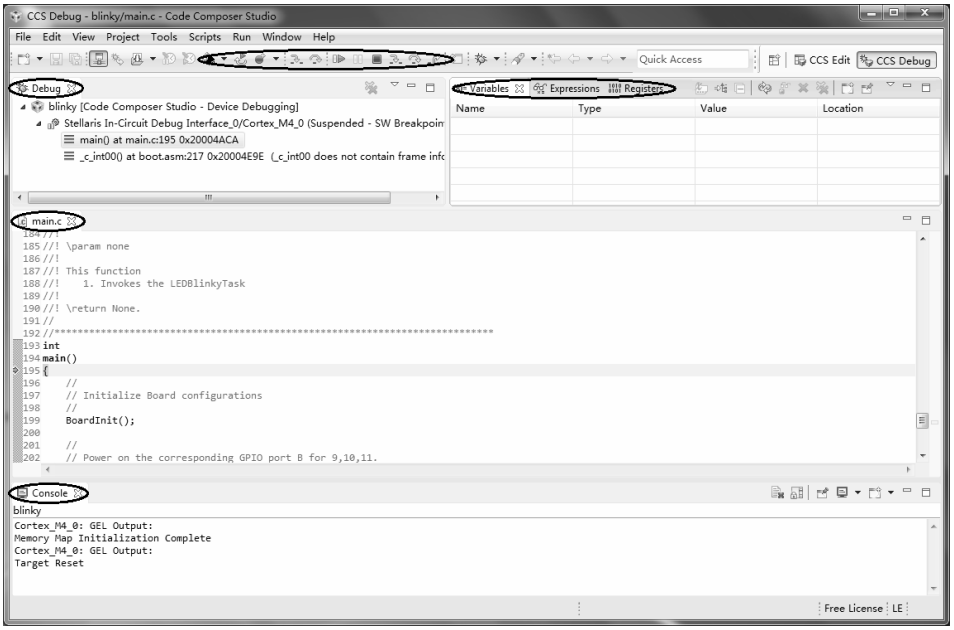


图 2.20　CCS 调试界面

CCS 调试界面主要包括调试工具栏、调试（Debug）视图、变量（Variables）/表达式（Expressions）/寄存器（Registers）视图、主程序（main.c）视图和控制台（Console）视图等，其中调试工具栏如图 2.21 所示。



图 2.21　CCS 调试工具栏

（5）单击“Run”（运行）→“Resume”（运行）或单击调试工具栏中的“Resume”（运行）按钮或按“F8”键，运行程序，开发板上的 3 个用户 LED 首先全部熄灭，然后红 LED、黄 LED 和绿 LED 依次循环闪烁。

（6）单击调试工具栏中的“Terminate”（结束）按钮，结束调试返回编辑界面。

CCS 调试界面的具体使用参见后续章节的详细介绍。

3．重建 SDK 驱动程序库

为了使 CCS 能在新的安装路径中找到 SDK 驱动程序，需要重建 SDK 驱动程序库，包括外设驱动程序库、网络驱动程序库、操作系统程序库和 TI-RTOS 配置等。

重建 SDK 驱动程序库的步骤如下。

（1）参考图 2.15 导入 driverlib、oslib、simplelink、ti_rtos_config 驱动程序库项目。

注意：在导入上述项目时，不要选择“Copy projects into workspace”（复制项目到工作空间）选项，否则编译程序将无法找到新生成的库文件。

（2）右击“Project Explorer”（项目浏览器）中的 driverlib 项目，在弹出菜单中单击“Properties”（属性），打开“属性”对话框，选择“Compiler version”（编译器版本）为 TI v5.2.2，如图 2.17 所示。

（3）右击“Project Explorer”（项目浏览器）中的 driverlib 项目，在弹出菜单中单击“Build Project”（生成项目）重建外设驱动程序库。

（4）重复步骤（2）和步骤（3），用相同的方法重建 SimpleLink。

注意：SimpleLink 有 4 种配置（Configuration）：NON_OS、NON_OS_PM、OS 和 PM_framework，这 4 种配置需要分别进行设置和重建，参见步骤（6）。

（5）重建 ti_rtos_config 项目时，除在“属性”对话框“main”标签选择“Compiler version”（编译器版本）为“TI v5.2.2”外，还需要在“RTSC”标签选择最新 XDCtools 和 TI-RTOS 版本，不选择 bios 和 uia，确认“Target”（目标）是“ti.targets.arm.elf.M4”，选择“Platform”（平台）为“ti.platforms.simplelink:CC3200”，如图 2.22 所示。

（6）重建 oslib 项目时，需首先在“属性”对话框中选择“Configuration”（配置）为 ti_rtos，单击“Manage Configurations…”（管理配置）按钮打开管理配置对话框进行配置管理，如图 2.23 所示，然后在“main”标签选择“Compiler version”（编译器版本）为“TI v5.2.2”。

注意：oslib 项目使用了 ti_rtos_config 项目，因此必须先重建 ti_rtos_config 项目，再重建 oslib 项目。

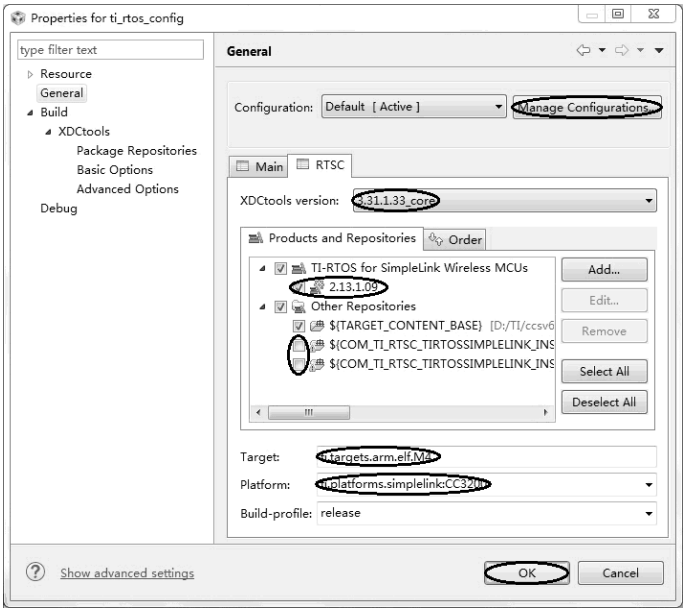


图 2.22　ti_rtos_config 项目属性对话框

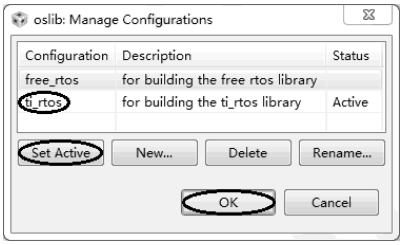


图 2.23　oslib 项目管理配置对话框

2.1.3　实用程序

CC3200 的实用工具主要有 PinMux 和 UniFlash。

1．PinMux

PinMux 为通用引脚的个性化选择提供便利的接口，PinMux 根据选择信息生成的源文件可直接包含在项目中。

PinMux v3.0 的安装文件是“pinmux-3.0.334-setup.exe”，PinMux v3.0 在 Windows 7 中的安装步骤如下。

（1）双击 pinmux-3.0.334-setup.exe，显示如图 2.24 所示的安装向导。



图 2.24　PinMux 安装向导

（2）单击“Next”（下一步）按钮，显示如图 2.25 所示的许可协议。



图 2.25　PinMux 许可协议

（3）阅读许可协议，选中“I accept the agreement”（我接受此协议），单击“Next”（下一步）按钮，显示如图 2.26 所示的安装目录。

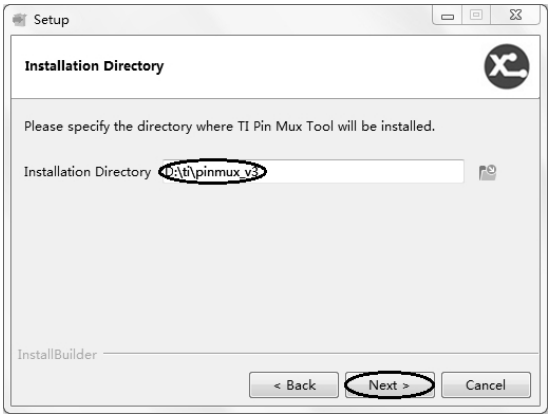


图 2.26　PinMux 安装目录

（4）接受默认安装目录或修改安装目录，单击“Next”（下一步）按钮两次，开始安装。安装完成后，显示如图 2.27 所示的安装完成对话框。

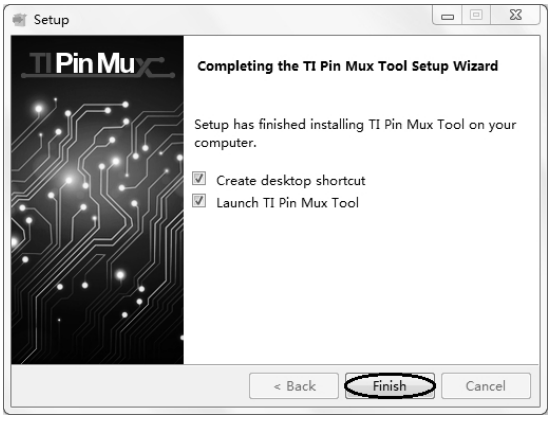


图 2.27　PinMux 安装完成

（5）单击“Finish”（完成）按钮，创建桌面快捷方式，并执行 PinMux。

PinMux 的使用请参考 3.1.4 节。

2．UniFlash

UniFlash 用于将固件、应用程序和证书等编程到串行闪存。UniFlash v3.2 的安装文件是“uniflash_cc3xxx_setup_3.2.0.00123.exe”，在 Windows 7 中的安装步骤如下。

（1）双击 uniflash_cc3xxx_setup_3.2.0.00123.exe，显示如图 2.28 所示的安装向导。

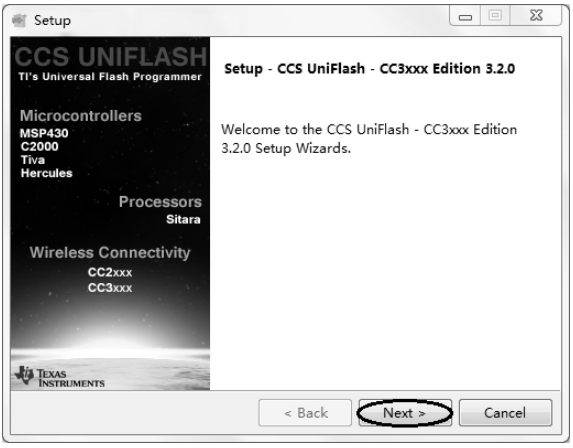


图 2.28　UniFlash 安装向导

（2）单击“Next”（下一步）按钮，显示如图 2.29 所示的许可协议。



图 2.29　UniFlash 许可协议

（3）阅读许可协议，选中“I accept the agreement”（我接受此协议），单击“Next”（下一步）按钮，显示如图 2.30 所示的安装目录。

（4）接受默认安装目录或修改安装目录，单击“Next”（下一步）按钮两次，开始安装。安装完成后，显示如图 2.31 所示的安装完成对话框。

（5）单击“Finish”（完成），创建快捷方式，并执行 UniFlash。

UniFlash 的使用请参考 2.2.2 节和 8.2 节。

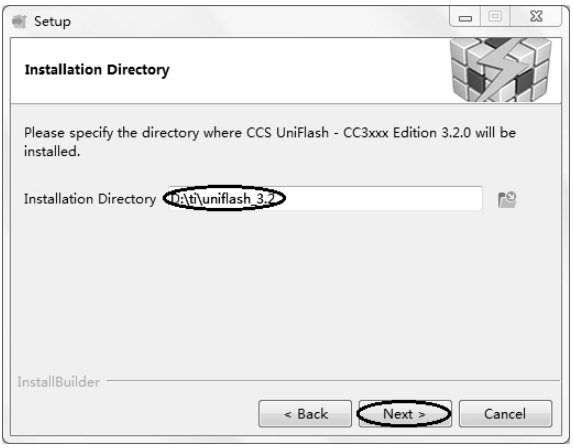


图 2.30　UniFlash 安装目录

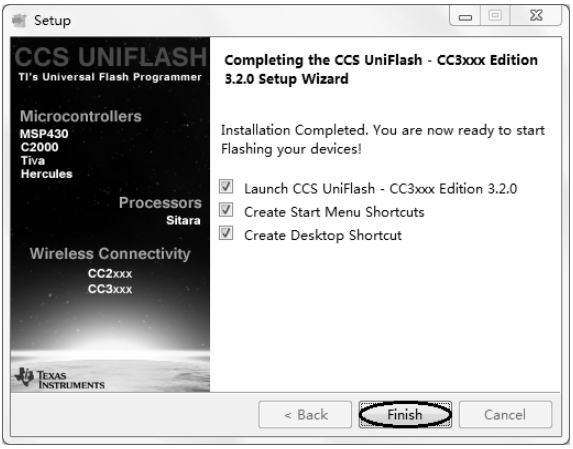


图 2.31　UniFlash 安装完成

2.1.4　支持工具

CC3200 的支持工具主要有超级终端、Iperf 和 SmartConfig 等。

1．超级终端

超级终端是一个通用的串行交互软件，可以通过串口、调制解调器或以太网连接到其他计算机、Telnet 站点、公告板系统（BBS）、联机服务和主机，以及嵌入式系统等。

Windows XP 自带超级终端，而 Windows 7 中没有超级终端，可以将 Windows XP 自带的超级终端复制到 Windows 7 中使用，也可以使用其他的串口通信工具。

Windows XP 自带的超级终端包含“C:\Program Files\Windows NT\hypertrm.exe”和“C:\WINDOWS\system32\hypertrm.dll”两个文件。

下面以 Tera Term 为例，介绍 Windows 7 中超级终端的安装和设置。Tera Term 是一款简单但功能强大的开源终端仿真器，主要特性如下。

- 支持 Telnet 和 SSH2。

- 内置 Web 服务。
- Tera Term 宏语言。
- 免费。

Tera Term v4.85 的安装文件是“teraterm485.exe”，Tera Term v4.85 在 Windows 7 中的安装步骤如下。

（1）双击 teraterm485.exe，显示如图 2.32 所示的安装向导。

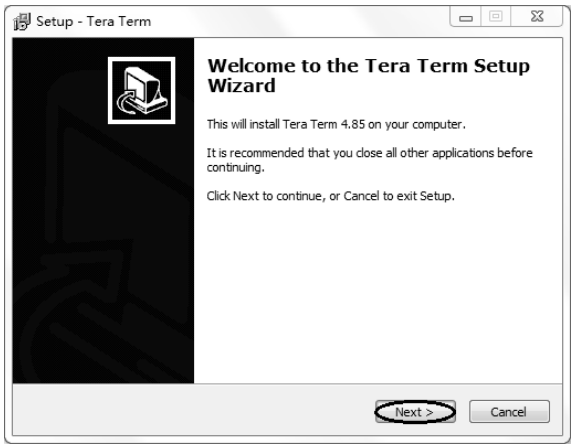


图 2.32　Tera Term 安装向导

（2）单击“Next”（下一步）按钮，显示如图 2.33 所示的许可协议。

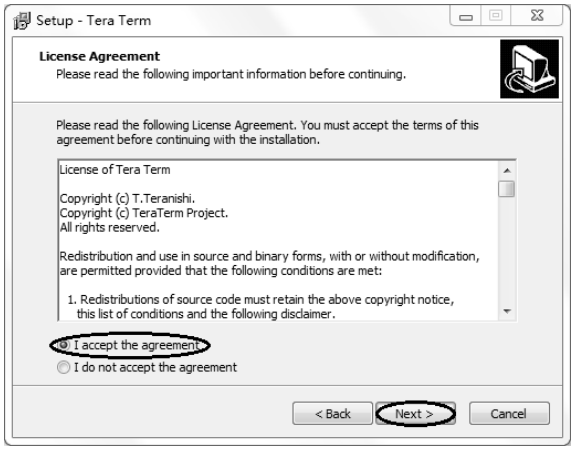


图 2.33　Tera Term 许可协议

（3）阅读许可协议，单击“I accept the agreement”（我接受此协议），单击“Next”（下一步）按钮，显示如图 2.34 所示的安装目录。

（4）接受默认安装目录或修改安装目录，单击“Next”（下一步）按钮，显示如图 2.35 所示的选择组件对话框。

（5）接受默认选择或选择其他安装方式，单击“Next”（下一步）按钮，显示如图 2.36 所示的选择语言对话框。

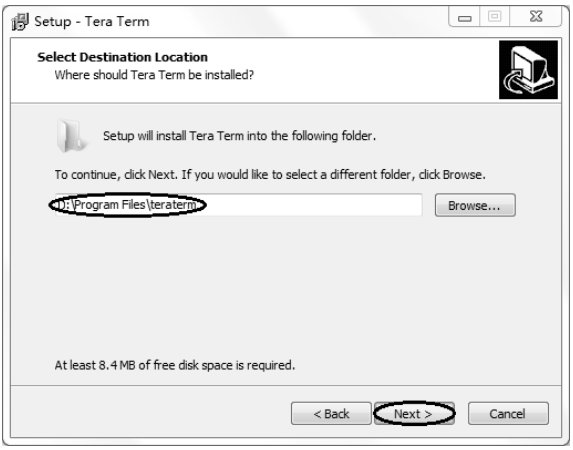


图 2.34　Tera Term 安装目录

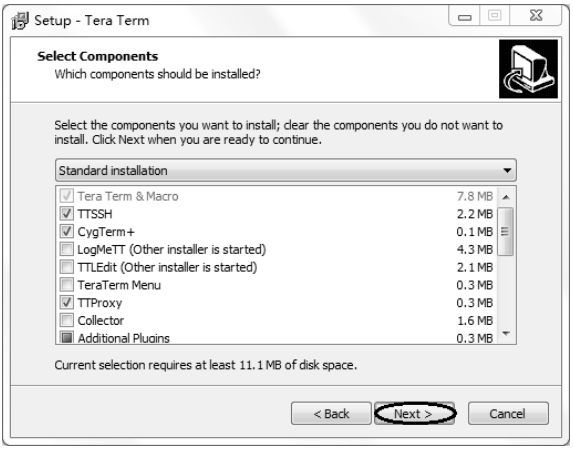


图 2.35　Tera Term 选择组件



图 2.36　Tera Term 选择语言

（6）接受默认选择或选择 Chinese(Simplified)（简体中文），单击“Next”（下一步）按钮，显示如图 2.37 所示的选择开始菜单文件夹对话框。

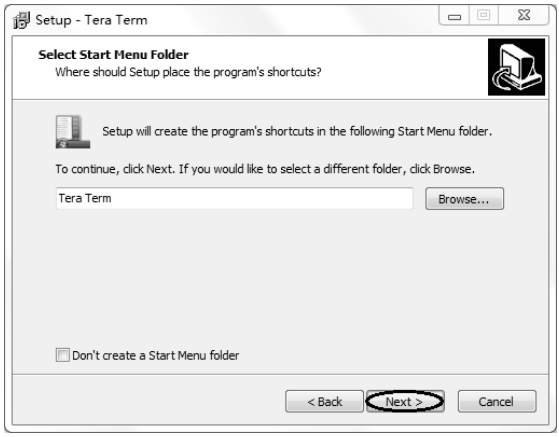


图 2.37　Tera Term 选择开始菜单文件夹

（7）单击“Next”（下一步）按钮接受默认选择，显示如图 2.38 所示的选择添加任务对话框。

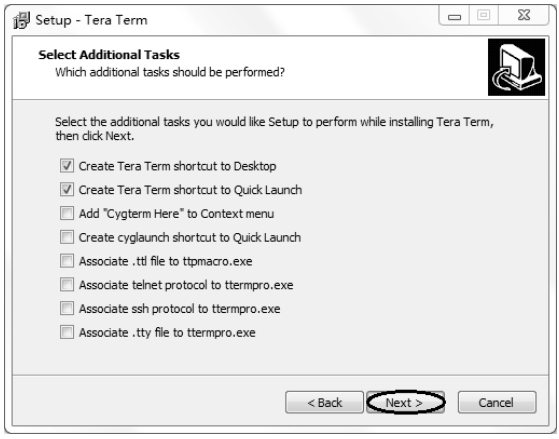


图 2.38　Tera Term 选择添加任务

（8）单击“Next”（下一步）按钮接受默认选择，显示如图 2.39 所示的准备安装对话框。

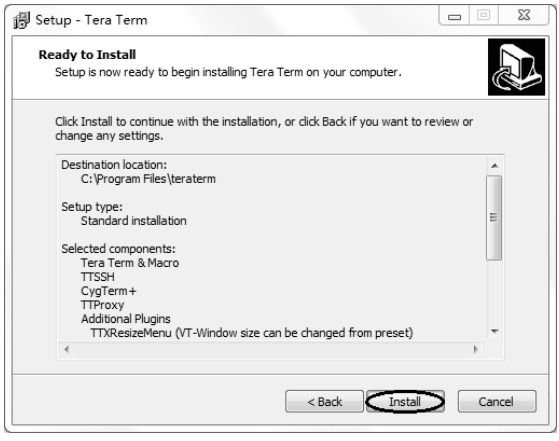


图 2.39　Tera Term 准备安装

（9）单击“Install”（安装）按钮，开始安装 Tera Term，安装完成后，显示如图 2.40 所示的安装完成对话框。

（10）选择“Launch Tera Term”（执行 Tera Term），单击“Finish”（完成）按钮，完成安装并执行 Tera Term，显示如图 2.41 所示的新建连接对话框。



图 2.40　Tera Term 安装完成



图 2.41　Tera Term 新建连接

（11）单击“串口”，如果已将 CC3200 LaunchPad 等硬件开发平台连接到 PC，并且正确安装了 FTDI 驱动程序，则“端口”下拉列表中会出现 USB-UART 转换后的串行端口（如 COM22），选择该串行端口，单击“确定”按钮，显示 Tera Term 窗口。

（12）单击 Tera Term“设置”菜单下的“窗口”菜单项，打开如图 2.42 所示的窗口设置对话框。



图 2.42　Tera Term“窗口设置”对话框

（13）选择“光标形状”为“横线”，单击“翻转”按钮，将黑底白字翻转为白底黑字，单击“确定”按钮关闭“窗口设置”对话框。

（14）单击 Tera Term“设置”菜单下的“字体”菜单项，打开如图 2.43 所示的字体对话框。

（15）选择“字体”为 Consolas，单击“确定”按钮关闭字体对话框。

（16）单击 Tera Term“设置”菜单下的“串口”菜单项，打开如图 2.44 所示的串口设置的对话框。

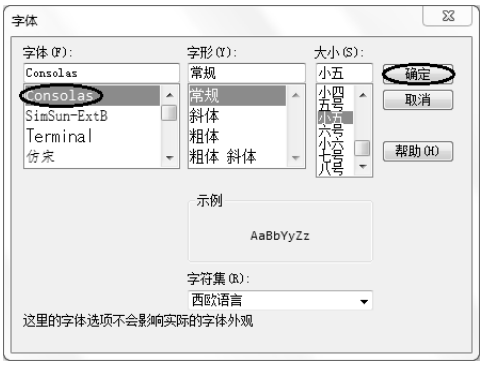


图 2.43　Tera Term“字体”对话框



图 2.44　Tera Term“串口设置”对话框

（17）选择“波特率”为 115200，单击“确定”按钮关闭串口设置对话框。

（18）单击 Tera Term“设置”菜单下的“保存配置”菜单项，打开如图 2.45 所示的保存配置对话框。

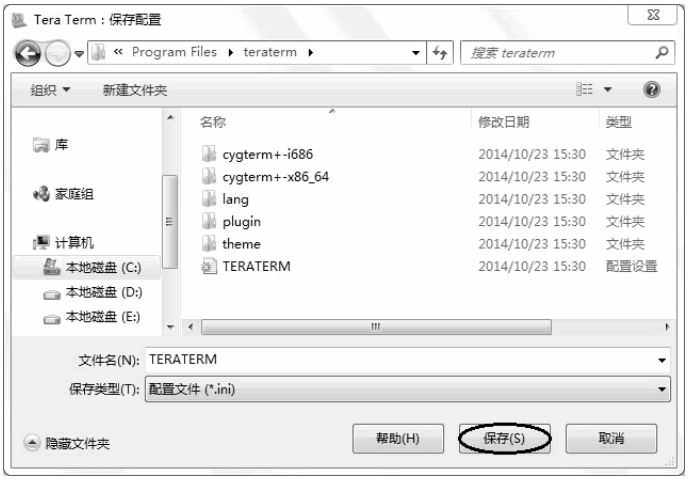


图 2.45　Tera Term“保存配置”对话框

（19）单击“保存”按钮保存配置，关闭保存配置对话框，完成 Tera Term 设置。

2．Iperf

Iperf 是一个网络性能测试工具，可以测试最大 TCP 和 UDP 带宽性能。Iperf 具有多种参数，可以根据需要调整。Iperf 可以报告带宽、延迟抖动和数据包丢失等。

Iperf有UNIX/Linux版、Windows版和Android版等多种版本。为了测试方便，可在Android手机上安装 Android 版，Android 版的安装文件是 Iperf.apk，其安装和一般 Android 手机软件的安装方法相同。

3．SmartConfig

SmartConfig 技术是 TI 独特的一步式 Wi-Fi 智能配置技术，允许多个 CC3200 设备快速高效地连接到 Wi-Fi 网络。

考虑到 CC3200 设备通常没有用于输入 Wi-Fi 网络名称和密码的显示屏或键盘，Smart Config 技术为用户提供了将基于 CC3200 的设备轻松连接到接入点的能力。

SmartConfig Android 版的安装文件是 SmartConfigCC3X.apk，其安装和一般 Android 手机软件的安装方法相同。

2.1.5 ROM 服务

CC3200 ROM 中包含启动引导程序和外设驱动库。启动引导程序允许用户更新串行闪存中的应用程序映像和其他用户文件，同时还负责将串行闪存中的应用程序加载到应用 MCU 的 RAM 中。在 ROM 中提供外设驱动库可以减少应用程序占用 RAM 的容量。

1．启动引导程序

启动引导程序驻留在应用 MCU 的 ROM 中，包括更新/下载和自举两项功能。

- 更新/下载：下载应用程序映像到 CC3200，下载功能只有在 CC3200 的 UART 装载模式（LDfrUART：SoP[2:0]=100）时才能使用。
- 自举：负责扫描串行闪存中的有效应用程序映像，将有效应用程序映像加载到应用 MCU 的 RAM 中，并将运行控制交给应用程序。

（1）启动引导模式。CC3200 有 3 个上电感知（SoP）引脚，可以设置下列三种操作模式。

① Fn4WJ（SoP[2:0]=000）：4 线 JTAG 功能开发模式，4 个 JTAG 引脚 TDI、TDO、TMS 和 TCK 用于调试器连接，启动时将应用程序映像从串行闪存加载到 RAM 中运行。

② Fn2WJ（SoP[2:0]=001）：2 线 SWD 功能开发模式，2 个 SWD 引脚 TMS 和 TCK 用于调试器连接，启动时将应用程序映像从串行闪存加载到 RAM 中运行。

③ LDfrUART（SoP[2:0]=100）：UART 下载模式，在开发和 OEM 装配时通过 UART 将应用程序映像下载到串行闪存。

与启动引导相关的是 Fn4WJ（SoP[2:0]=000：4 线 JTAG 功能开发模式）和 LDfrUART（SoP[2:0]=100：UART 下载模式）。

（2）启动引导程序和用户应用程序共享 MCU RAM。在 UART 下载模式，启动引导程序需要内存资源，这些资源来自 MCU RAM，启动引导程序使用的 RAM 容量是 16 KB。对于 256 KB 的 MCU RAM，用户应用程序映像需要限制在 240 KB，因此开发者需要关注以下两个关键点。

① MCU RAM 的地址范围 0x2000 0000～0x2000 3FFF（16 KB）被启动引导程序和用户应用程序共享。当启动引导程序将应用程序映像从串行闪存加载到 RAM 时，该区域被启动引导程序独占，一旦启动引导程序将运行控制交给应用程序，该区域将作为应用程序的数据段使用。

② MCU RAM 的地址范围 0x2000 4000～0x2003 FFFF（240 KB）被用户应用程序独占，用户应用程序映像总是被加载到 0x2000 4000 开始的该区域。

2．外设驱动库

应用 MCU ROM 中的外设驱动程序可以被用户应用程序调用，外设驱动程序的源代码可在 SDK 中使用，开发者可以选择生成从 RAM 直接使用设备驱动程序的应用程序。

ROM API 使用“重定向”方法进行调用，以允许未来扩展时 ROM 中访问函数的位置向后兼容，未来版本的 API 位置可能改变，但 API 表不会变。

ROM 中的主子两级表解决了 API 的访问入口问题，主表中每个外设对应一个指向各自子表的指针，子表中每个相关外设的 API 对应一个指向各自 API 的指针。

主表存放在 ROM 的开始地址为 0x0000 040C，部分内容如表 2.6 所示，其中中断子表的部分内容如表 2.7 所示。

表 2.6　ROM API 主表部分内容

序　号	说　明	序　号	说　明
[0]	保留	[1]	ROM_UARTTABLE 指针
[2]	ROM_TIMERTABLE 指针	[3]	ROM_WATCHDOGTABLE 指针
[4]	ROM_INTERRUPTTABLE 指针	[5]	ROM_UDMATABLE 指针
[6]	ROM_PRCMTABLE 指针	…	…

表 2.7　ROM API 中断子表部分内容

序　号	说　明	序　号	说　明
[0]	ROM_IntEnable 指针	[1]	ROM_IntMasterEnable 指针
[2]	ROM_IntMasterDisable 指针	…	…

主表和各子表的详细内容参见“driverlib\rom.h”文件。

（1）用 ROM API 替代 RAM API 的步骤如下。

① 所有使用 ROM API 的.c 文件都应该包含下列头文件。

```
#include "rom.h"。
#include "rom_map.h"。
```

② 所有项目文件应该添加全局预处理定义“TARGET_IS_CC3200”。

③ 所有 API 调用的 API 名称前应该添加“MAP_”。

④ 重建所有相关项目。

（2）用 RAM API 修补 ROM API 的步骤如下。

① 在“\driverlib\rom_patch.h”文件中为所有需要修补的 API 添加定义。

② 重建所有相关项目。

（3）重新用 RAM API 连接用户应用程序的步骤如下。

① 从所有使用 API 的项目文件中删除全局预处理定义“TARGET_IS_CC3200”。

② 重建所有相关项目。

2.2 硬件开发环境

CC3200 的硬件开发环境可以使用 TI 的 CC3200 LaunchPad，也可以使用其他兼容硬件开发环境。CC3200 LaunchPad 是 TI 开发的 CC3200 硬件开发平台，使用 FTDI 器件实现板载调试和 UART-USB 转换，并且包含温度传感器和加速度传感器提供开包即用体验。

CC3200 LaunchPad 具有下列特性。

- CC3200 Wi-Fi MCU，支持站点和接入点模式，支持 SmartConfig 和 WPS 网络配置。
- 板载闪存（1 MB）可通过 USB 在 PC 上使用 UniFlash 进行更新。
- 板载 FTDI 器件可实现 JTAG 调试和 UART-USB 转换。
- 板载温度传感器和加速度传感器可提供开包即用体验。
- 微型 USB 连接器用于供电、调试和 UART 通信。
- 支持 4 线 JTAG 和 2 线 SWD。
- 2 个 20 引脚连接器可实现与其他 LaunchPad 兼容设备的连接。
- 2 个按键和 3 个 LED 用于用户交互。
- 可通过修改电容器选择板载天线和外接天线。
- 可由 2 节 AA 碱性电池供电运行。

CC3200 LaunchPad 的功能框图、实物图和连接关系参见附录 E。

2.2.1 TI CC3200 LaunchPad 开包即用体验

CC3200 LaunchPad 预装了 out_of_box 项目，利用板载温度传感器和加速度传感器，以及 LED 可提供开包即用体验，体验步骤如下（out_of_box 项目的详细分析请参考 8.2 节）。

（1）确认 CC3200 LaunchPad 的 J2（1-2）、J3（1-2）、J6（2-3）、J7（2-3）、J8（1-2）、J9（1-2）、J10（1-2）、J11（1-2）、J12（1-2）、J13（1-2）和 J22.P1（1-2）用短路块连接，参见图 E.2。

警告：用短路块连接 J22.P1（1-2）时要特别注意，不要误将短路块连接到 J22.P3（1-2）上，否则会造成电源短路（J22.P3.1 是 5 V 电源，J22.P3.2 是 GND）!

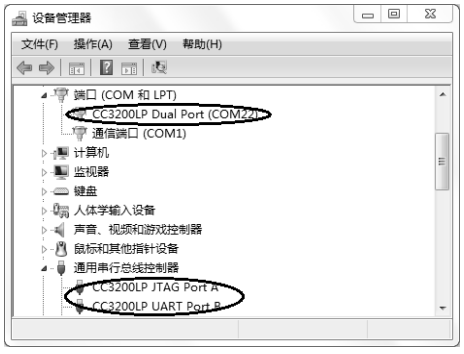


图 2.46 设备准备就绪

为了防止误连接，可以将 J22.P3.1 往外掰弯。

将 CC3200 LaunchPad 通过 USB 线与 PC 连接，CC3200 LaunchPad 上的 D1（黄 LED，复位指示）、D4（红 LED，电源指示）、D5（绿 LED）和 D6（黄 LED）亮，D7（红 LED）闪烁 3 次然后熄灭（如果 D7 不闪烁，尝试按一下 CC3200 LaunchPad 上的 RESET 按键），CC3200 LaunchPad 进入接入点模式。

同时，PC 自动加载 FTDI 驱动程序，加载完成后在设备管理器中可以看到图 2.46 所示内容，从图

中可以看出：FTDI 的端口 A 用于 JTAG 接口，端口 B 用于 UART 接口，UART 的接口号是 COM22（实际情况可能不同）。

注意：记住 UART 的接口号 COM22，UniFlash 中要用！

（2）在手机（或平板电脑或笔记本电脑）上打开 Wi-Fi，连接 mysimplelink-054D6C（实际情况可能不同）。

（3）在手机（或平板电脑或笔记本电脑）上打开网页浏览器，输入“mysimplelink.net”，显示如图 2.47 所示的欢迎界面。

（4）单击“Demos”（演示）标签，显示如图 2.48 所示的演示界面，在演示界面的下方有 4 个应用演示按钮，分别为家庭自动化、家用电器、安全保障和智能能源。

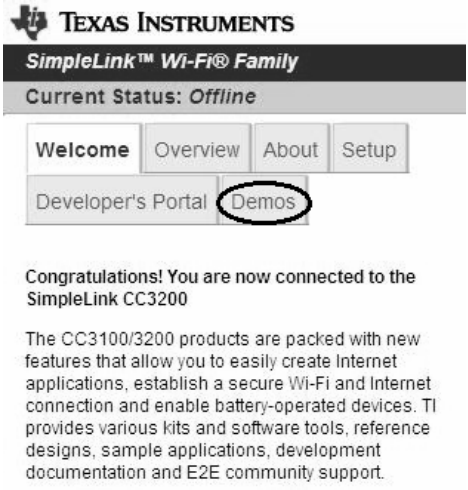


图 2.47　欢迎界面

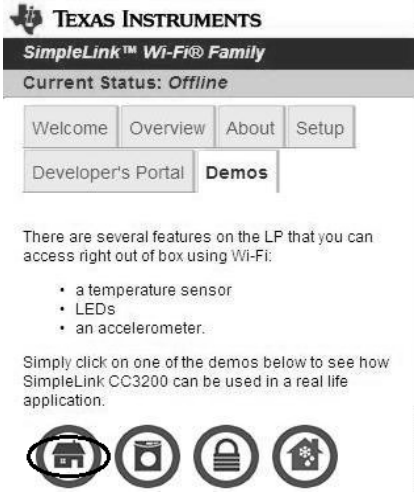


图 2.48　演示界面

（5）单击演示界面左下角的家庭自动化演示按钮，显示如图 2.49 所示的家庭自动化界面 1。

（6）单击左边的开关，开关打开（On），草地左下角的喷头间歇喷水，如图 2.50 所示，同时 CC3200 LaunchPad 上的 D7（红 LED）闪烁。

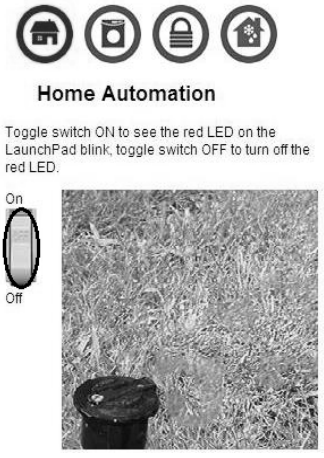


图 2.49　家庭自动化界面 1



图 2.50　家庭自动化界面 2

（7）单击家用电器按钮，显示如图 2.51 所示的家用电器界面 1。

（8）用手触摸 CC3200 LaunchPad 上的温度传感器，水温（Water Temp）发生变化；拿起 CC3200 LaunchPad，烘干机（Dryer）从停止（Stopped）变为运行（Running），如图 2.52 所示。



图 2.51　家用电器界面 1



图 2.52　家用电器界面 2

（9）单击安全保障按钮，显示如图 2.53 所示的安全保障界面 1。

（10）单击左边的开关，开关打开（On），拿起 CC3200 LaunchPad，系统报警，如图 2.54 所示。



图 2.53　安全保障界面 1



图 2.54　安全保障界面 2

（11）单击智能能源按钮，显示如图 2.55 所示的智能能源界面。

（12）单击箭头可以减小或增大所需温度（Desired Temperature），用手触摸 CC3200 LaunchPad 上的温度传感器，当前温度（Current Temperature）发生变化。

所需温度低于当前温度时 CC3200 LaunchPad 上的 D7（红 LED）闪烁，高于时 D7（红 LED）常亮，相等时 D7（红 LED）熄灭。

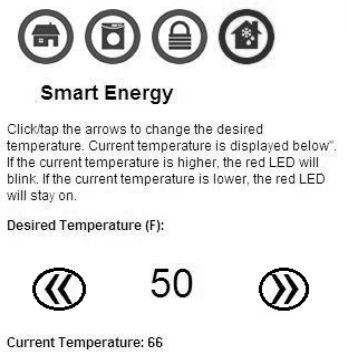


图 2.55　智能能源界面

2.2.2　服务包更新

如果 CC3200 LaunchPad 上服务包是 SDK v0.5.1/v0.5.2 或 v1.0.0，需要用 UniFlash 更新到 SDK v1.1.0，SDK v1.1.0 服务包的安装文件是“CC31xx_CC32xx_ServicePack-1.0.0.10.0-windows-installer.exe”，执行上述文件将服务包安装到默认目录，其中包含“servicepack_1.0.0.10.0.bin”服务包文件。服务包更新的具体步骤如下。

（1）拔掉 CC3200 LaunchPad 上连接 J22.P1（1-2）的短路块，将 J15（1-2）用短路块连接，并确认 J2（1-2）、J3（1-2）、J6（2-3）、J7（2-3）、J8（1-2）、J9（1-2）、J10（1-2）、J11（1-2）、J12（1-2）和 J13（1-2）用短路块连接。

（2）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（3）执行 CCS UniFlash-CC3xxx Edition 3.2.0（D:\TI\uniflash_3.2\eclipse\uniflash.exe），显示如图 2.56 所示的 CCS UniFlash 窗口。

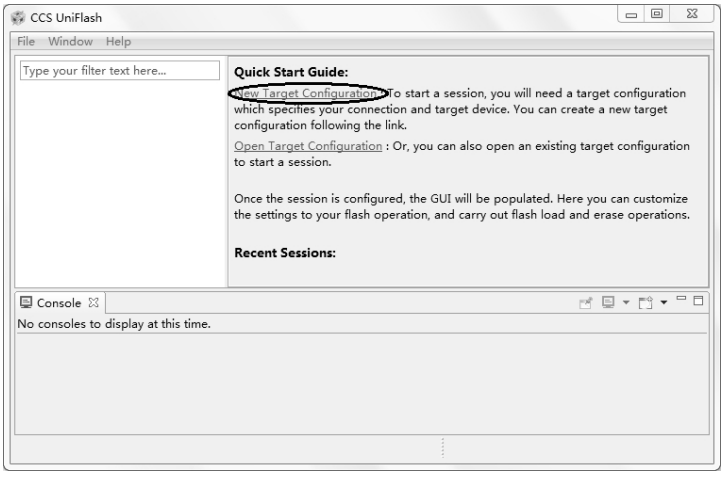


图 2.56　CCS UniFlash 窗口

（4）单击“File”（文件）菜单下的“New Configuratiom”（新建配置）菜单项或单击“Quick Start Guide”（快速开始指南）下的“New Target Configuratiom”（新建目标配置）打开如图 2.57 所示的新建配置对话框。

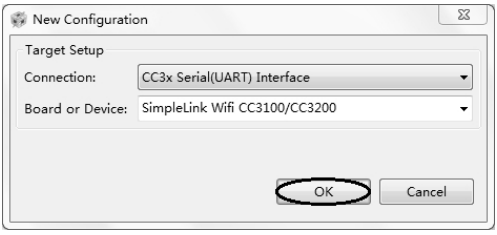


图 2.57　UniFlash“新建配置”对话框

（5）单击“OK”按钮接收默认目标设置，显示如图 2.58 所示的新建配置窗口。

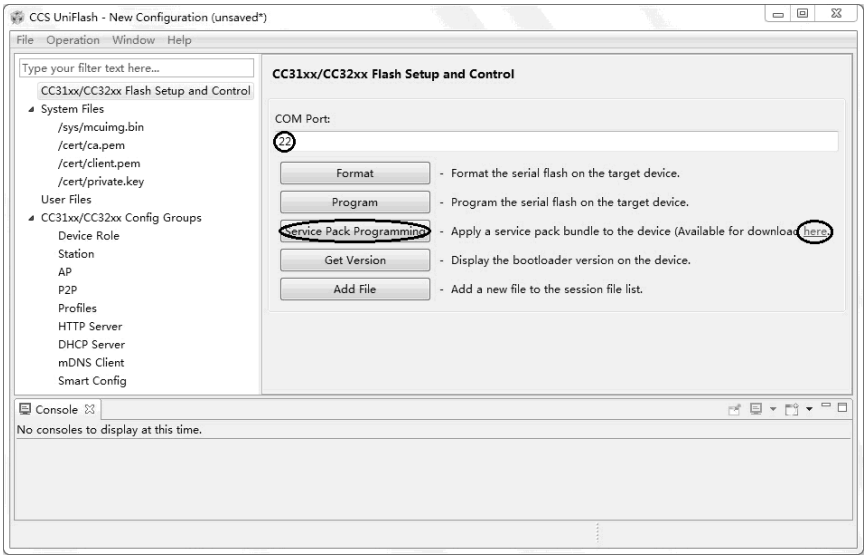


图 2.58　UniFlash 新建配置窗口

（6a）在“COM Port”（COM 接口）下输入图 2.46 中显示的 UART 接口号“22”，单击“Service Pack Programming”（服务包编程）按钮打开 servicepack_1.0.0.10.0.bin，开始更新服务包，Console（控制台）下显示更新过程。

注意：如果编程过程中止，控制台显示“please restart the device”（请重启设备），则需按一下开发板上的复位键重新启动设备。

单击“Service Pack Programming”（服务包编程）按钮右边的“here”（这里）链接可从 TI 官网上下载最新的服务包。

UniFlash 除了可以更新服务包外，还可以将用户应用程序映像编程到串行闪存中，具体步骤和更新服务包相似，不同的是：

（6b）输入 UART 接口号后，单击左侧“System Files”（系统文件）下的“/sys/mcuimg.bin”，再单击右侧的“Browse”（浏览）按钮，打开需要编程的应用程序映像文件，并选中“Erase”（擦除）和“Update”（更新），如图 2.59 所示。

（7）单击左侧的“CC31x Flash Setup and Control”，再单击右侧的“Program”（编程）按钮，将应用程序二进制映像文件编程到串行闪存中。

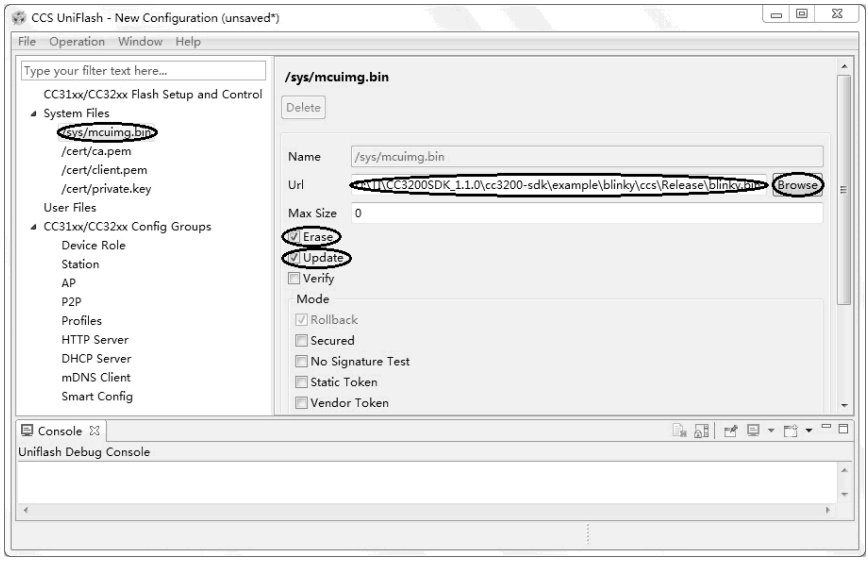


图 2.59　UniFlash 选择编程文件

（8）编程完成后，拔掉 CC3200 LaunchPad 上连接 J15（1-2）的短路块，按一下“RESET”（复位）按键 SW1，开始执行用户程序。

和前面的版本相比，UniFlash v3.2.0.00123 除了包含“System Files”（系统文件）外，还包含“User Files”（用户文件）和“CC31xx/CC322xx Config Groups”（CC31xx/CC322xx 配置组），特别是 CC31xx/CC322xx 配置组的设置，极大地方便了 Wi-Fi 的配置。

CC31xx/CC322xx 配置组包含下列内容。

- Device Role：设备角色，包括站点、接入点和点对点。
- Station：站点，包括 MAC 地址、发射功率、电源策略和网络应用等。
- AP：接入点，包括 MAC 地址、名称、安全类型、密码和网络应用等。
- P2P：点对点，包括 MAC 地址、发射功率、电源策略和网络应用等。
- Profiles：配置文件，包括名称、优先级和安全类型等。
- HTTP Server：HTTP 服务器，包括端口号等。
- DHCP Server：DHCP 服务器，包括租用时间和起止 IP 地址。
- mDNS Client：mDNS 客户端，包括服务名称等。
- Smart Config：智能配置，包括默认组密码等。

第 3 章

片内外设接口应用

CC3200 支持下列片内外设接口。

- 通用输入输出接口（GPIO）。
- 串行异步收发器接口（UART）。
- 串行外设接口（SPI）。
- 内部集成电路总线接口（I2C）。
- 多通道音频串行接口（McASP）。
- SD/MMC 接口。
- 并行摄像头接口。

本章主要在分析片内接口示例程序的基础上，介绍片内接口的使用，并对示例程序进行修改，完善示例程序的功能。

3.1 GPIO 接口应用

CC3200 的所有数字引脚和部分模拟引脚可作为通用输入/输出（GPIO）使用，GPIO 分为 4 组（GPIOA0～3），每组 8 个（具体引脚分配参见表 D.1）。

- GPIOA0：GPIO_00～GPIO_07，对应 GPIOA0.0～GPIOA0.7。
- GPIOA1：GPIO_08～GPIO_15，对应 GPIOA1.0～GPIOA1.7。
- GPIOA2：GPIO_16～GPIO_23，对应 GPIOA2.0～GPIOA2.7。
- GPIOA3：GPIO_24～GPIO_31，对应 GPIOA3.0～GPIOA3.7。

CC3200 GPIO 具有下列特性。

- 根据功能引脚的配置不同，GPIO 的数量多达 27 个。
- 所有 GPIO 引脚具有中断能力。
 - ✧ 电平有效或边沿有效。
 - ✧ 上升沿或下降沿。
 - ✧ 可选择中断屏蔽。
- 可以用于触发 DMA 操作。
- 可选择作为唤醒源。
- 可编程引脚配置。
 - ✧ 内部 10 μA 上拉或下拉。
 - ✧ 2 mA、4 mA、6 mA、8 mA、10 mA、12 mA、14 mA 可配置驱动能力。

✧ 开漏模式。

- 可通过高速内部总线矩阵进行 GPIO 寄存器读操作。

3.1.1 GPIO 示例程序分析

GPIO 示例程序在“example\blinky”目录中，Blinky 项目 CC3200 LaunchPad 电路如图 3.1 所示。

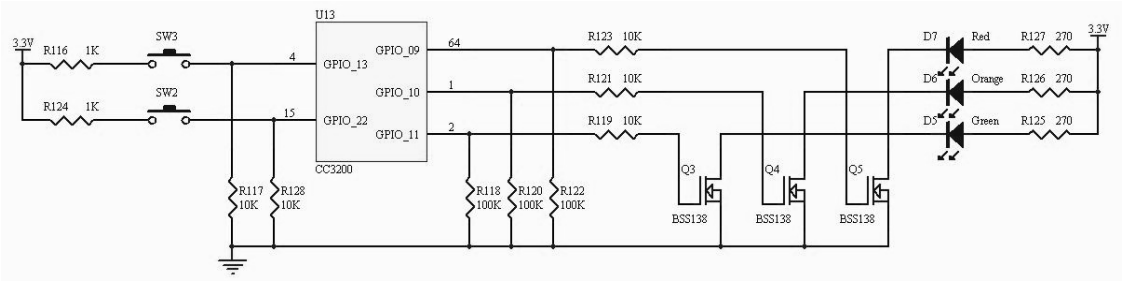


图 3.1 Blinky 项目 CC3200 LaunchPad 电路图

两个用户按键 SW2 和 SW3 分别通过 GPIO_22（PIN_15）和 GPIO_13（PIN_04）与 CC3200 相连，按键未按下时 GPIO 为低电平，按键按下时 GPIO 为高电平。

3 个用户 LED D5～D7 分别通过驱动电路和 GPIO_11（PIN_02）、GPIO_10（PIN_01）和 GPIO_09（PIN_64）与 CC3200 相连，GPIO 输出低电平时 LED 熄灭，GPIO 输出高电平时 LED 点亮。

（1）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（2）参考 2.1.2 节中的“CCS 使用”，将 Blinky 项目导入 CCS，在“Project Explorer”（项目浏览器）视图中可以看出，Blinky 项目包含下列 4 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO 引脚配置程序文件。
- gpio_if.c：GPIO 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Blinky 项目程序流程图如图 3.2 所示。

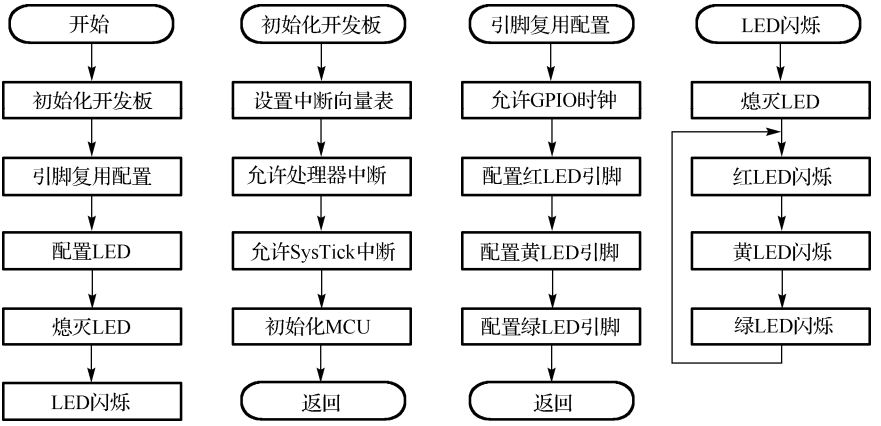


图 3.2 Blinky 项目程序流程图

（3）在 CCS 中单击“Project”（项目）→“Properties”（属性）或右击“Project Explorer”（项目浏览器）中的 Blinky 项目，在弹出菜单中单击“Properties”（属性），打开“属性”对话框，确认“Device”（器件）选择 CC3200，选择“Compiler version”（编译器版本）为 TI v5.2.2，参见图 2.17。

（4）单击“View”（视图）→“Target Configuration”（目标配置），打开“目标配置”视图，确认默认的目标配置为 CC3200.ccxml，如图 3.3 所示。

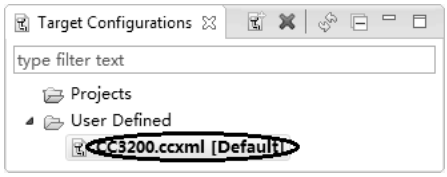


图 3.3　Blinky 项目“目标配置”视图

（5）在 CCS 中单击“Run”（运行）→“Debug”（调试）或按“F11”键，CCS 首先编译生成目标文件 blinky.out，然后将目标文件 blinky.out 下载到 CC3200 并进入调试界面。

（6）单击“Run”（运行）→“Resume”（运行）或单击调试工具栏中的“Resume”（运行）按钮或按“F8”键，运行程序，开发板上的 3 个用户 LED 首先全部熄灭，然后红 LED、黄 LED 和绿 LED 依次循环闪烁。

（7）单击调试工具栏中的“Suspend”（暂停）按钮暂停程序的运行。

（8）单击调试工具栏中的“Restart”（重新开始）按钮，程序停在 main()函数的开始处，如图 3.4 所示。

```
193 int
194 main()
195 {
196     //
197     // Initialize Board configurations
198     //
199     BoardInit();
200
201     //
202     // Power on the corresponding GPIO port B for 9,10,11.
203     // Set up the GPIO lines to mode 0 (GPIO)
204     //
205     PinMuxConfig();
206     GPIO_IF_LedConfigure(LED1|LED2|LED3);
207
208     GPIO_IF_LedOff(MCU_ALL_LED_IND);
209
210     //
211     // Start the LEDBlinkyRoutine
212     //
213     LEDBlinkyRoutine();
214     return 0;
215 }
```

图 3.4　Blinky 项目 main()函数

注意：在程序的分析过程中，可随时单击“Restart”（重新开始）按钮复位程序。

下面详细介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- LED 操作。

1．初始化开发板 BoardInit()

单击调试工具栏中的“Step Into”（单步进入）按钮两次，进入 BoardInit()函数，如图 3.5 所示。

```
157 static void
158 BoardInit(void)
159 {
160 /* In case of TI-RTOS vector table is initialize by OS itself */
161 #ifndef USE_TIRTOS
162     //
163     // Set vector table base
164     //
165 #if defined(ccs)
166     MAP_IntVTableBaseSet((unsigned long)&g_pfnVectors[0]);
167 #endif
168 #if defined(ewarm)
169     MAP_IntVTableBaseSet((unsigned long)&__vector_table);
170 #endif
171 #endif
172
173     //
174     // Enable Processor
175     //
176     MAP_IntMasterEnable();
177     MAP_IntEnable(FAULT_SYSTICK);
178
179     PRCMCC3200MCUInit();
180 }
```

图 3.5　Blinky 项目 BoardInit()函数

从图 3.5 中可以看出，初始化开发板 BoardInit()包括下列内容。

- 设置中断向量表：IntVTableBaseSet()。
- 允许处理器中断：IntMasterEnable()。
- 允许中断：IntEnable()。
- 初始化 MCU：PRCMCC3200MCUInit()。

（1）设置中断向量表 IntVTableBaseSet()。

```
void IntVTableBaseSet(unsigned long ulVtableBase);
```

功能：设置中断向量表基地址。

参数说明：ulVtableBase，中断向量表基地址，中断向量表在 startup_ccs.c 中定义。

IntVTableBaseSet()的声明在 interrupt.h 中，代码在 interrupt.c 中。

（2）允许处理器中断 IntMasterEnable()。

```
tBoolean IntMasterEnable(void);
```

功能：允许处理器中断。

IntMasterEnable()调用 CPUcpsie()实现允许处理器中断操作，其声明在 interrupt.h 中，代码在 interrupt.c 中。

```
unsigned long CPUcpsie(void);
```

功能：允许 CPU 中断。

CPUcpsie()调用汇编语言程序段实现允许 CPU 中断操作，其声明在 cpu.h 中，代码在 cpu.c 中。

（3）允许中断 IntEnable()。

```
void IntEnable(unsigned long ulInterrupt);
```

功能：允许中断。

参数说明：ulInterrupt，中断号，FAULT_SYSTICK 是 SysTick 的中断号常量，中断号是 15，在 hw_ints.h 中定义。

IntEnable()的声明在 interrupt.h 中，代码在 interrupt.c 中。

（4）初始化 MCU PRCMCC3200MCUInit()。

```
void PRCMCC3200MCUInit(void);
```

功能：初始化 MCU。

加电启动或从低功耗休眠模式退出时，应用程序应该通过调用 PRCMCC3200MCUInit()配置 MCU 参数。PRCMCC3200MCUInit()的声明在 prcm.h 中，代码在 prcm.c 中。

单击调试工具栏中的“Step Return”（单步返回）按钮，运行上述函数，然后退出 BoardInit()函数，回到 main()函数的 PinMuxConfig()语句。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序由 PinMux 应用程序根据设计须自动生成，程序代码在 pinmux.c 或 rom_pin_mux_config.c 中，其中调用了相应的 PRCM、PIN 和 GPIO API 函数。

单击调试工具栏中的“Step Into”（单步进入）按钮，进入 PinMuxConfig()函数，如图 3.6 所示。

```
55 void
56 PinMuxConfig(void)
57 {
58     //
59     // Enable Peripheral Clocks
60     //
61     MAP_PRCMPeripheralClkEnable(PRCM_GPIOA1, PRCM_RUN_MODE_CLK);
62
63     //
64     // Configure PIN_64 for GPIOOutput
65     //
66     MAP_PinTypeGPIO(PIN_64, PIN_MODE_0, false);
67     MAP_GPIODirModeSet(GPIOA1_BASE, 0x2, GPIO_DIR_MODE_OUT);
68
69     //
70     // Configure PIN_01 for GPIOOutput
71     //
72     MAP_PinTypeGPIO(PIN_01, PIN_MODE_0, false);
73     MAP_GPIODirModeSet(GPIOA1_BASE, 0x4, GPIO_DIR_MODE_OUT);
74
75     //
76     // Configure PIN_02 for GPIOOutput
77     //
78     MAP_PinTypeGPIO(PIN_02, PIN_MODE_0, false);
79     MAP_GPIODirModeSet(GPIOA1_BASE, 0x8, GPIO_DIR_MODE_OUT);
80 }
```

图 3.6　Blinky 项目 PinMuxConfig()函数

从图中可以看出：GPIO 引脚复用配置包括下列内容。

- 允许 GPIO 时钟：PRCMPeripheralClkEnable()。
- 配置 GPIO 引脚：PinTypeGPIO()。
- 配置 GPIO 方向：GPIODirModeSet()。

（1）允许 GPIO 时钟 PRCMPeripheralClkEnable()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮两次，进入 PRCMPeripheralClkEnable()函数，如图 3.7 所示。

```
352 void
353 PRCMPeripheralClkEnable(unsigned long ulPeripheral, unsigned long ulClkFlags)
354 {
355   //
356   // Enable the specified peripheral clocks, Nothing to be done for PRCM_ADC
357   // as it is a dummy define for pinmux utility code generation
358   //
359   if(ulPeripheral != PRCM_ADC)
360   {
361     HWREG(ARCM_BASE + PRCM_PeriphRegsList[ulPeripheral].ulClkReg) |= ulClkFlags;
362   }
363   //
364   // Set the default clock for camera
365   //
366   if(ulPeripheral == PRCM_CAMERA)
367   {
368     HWREG(ARCM_BASE + APPS_RCM_O_CAMERA_CLK_GEN) = 0x0404;
369   }
370 }
```

图 3.7　Blinky 项目 PRCMPeripheralClkEnable()函数

从图 3.7 中可以看出，PRCMPeripheralClkEnable()等 API 函数通过硬件寄存器操作 HWREG()实现相应的功能。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：外设号，GPIOA0～3 的外设号是 6～9，在 prcm.h 中定义。
- ulClkFlags：时钟标志，设置值是 PRCM_RUN_MODE_CLK（运行模式时钟）。ulClkFlags 包括下列值。
 - ✧ PRCM_RUN_MODE_CLK：运行模式时钟，0x0000 0001。
 - ✧ PRCM_SLP_MODE_CLK：睡眠模式时钟，0x0000 0100。
 - ✧ PRCM_DSLP_MODE_CLK：深睡眠模式时钟，0x0001 0000。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

② 单击调试工具栏中的“Step Return”（单步返回）按钮，运行上述函数，然后退出函数，返回到 PinMuxConfig()函数的 MAP_PinTypeGPIO()语句。

（2）配置 GPIO 引脚 PinTypeGPIO()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 PinTypeGPIO()函数，如图 3.8 所示。

```
582 void PinTypeGPIO(unsigned long ulPin,unsigned long ulPinMode,tBoolean bOpenDrain)
583 {
584
585     //
586     // Set the pin for standard push-pull operation.
587     //
588     if(bOpenDrain)
589     {
590             PinConfigSet(ulPin, PIN_STRENGTH_2MA, PIN_TYPE_OD);
591     }
592     else
593     {
594             PinConfigSet(ulPin, PIN_STRENGTH_2MA, PIN_TYPE_STD);
595     }
596
597     //
598     // Set the pin to specified mode
599     //
600     PinModeSet(ulPin, ulPinMode);
601
602 }
```

图 3.8　Blinky 项目 PinTypeGPIO()函数

```
void PinTypeGPIO(unsigned long ulPin, unsigned long ulPinMode,
                 tBoolean bOpenDrain);
```

功能：配置 GPIO 引脚。

参数说明：

- ulPin：引脚号，PIN_01～64，在 pin.h 中定义。
- ulPinMode：引脚模式，PIN_MODE_0 为 GPIO 模式。
- bOpenDrain：开漏输出，true 表示是，false 表示否。

PinTypeGPIO()的声明在 pin.h 中，代码在 pin.c 中。PinTypeGPIO()通过调用 PinConfigSet()和 PinModeSet()实现引脚类型的配置。

```
void PinConfigSet(unsigned long ulPin, unsigned long ulPinStrength,
                  unsigned long ulPinType);
```

功能：设置引脚类型。

参数说明：

- ulPin：引脚号，PIN_01～64，在 pin.h 中定义。
- ulPinStrength：引脚强度，引脚的输出驱动电流（2 mA、4 mA 或 6 mA 等）。
- ulPinType：引脚类型，设置值是 PIN_TYPE_STD（标准输入输出）。ulPinType 主要包括下列值。
 - ✧ PIN_TYPE_STD：标准输入输出，0x0000 0000。
 - ✧ PIN_TYPE_OD：开漏输出，0x0000 0010。
 - ✧ PIN_TYPE_ANALOG：模拟引脚，0x1000 0000。

PinConfigSet()的声明在 pin.h 中，代码在 pin.c 中。

```
void PinModeSet(unsigned long ulPin, unsigned long ulPinMode);
```

功能：设置引脚模式。

参数说明：

- ulPin：引脚号，PIN_01～64，在 pin.h 中定义。
- ulPinMode：引脚模式，PIN_MODE_0 为 GPIO 模式。

PinModeSet()的声明在 pin.h 中，代码在 pin.c 中。

② 单击调试工具栏中的“Step Return”（单步返回）按钮，运行上述函数，然后退出函数，返回到 PinMuxConfig()函数的 MAP_GPIODirModeSet()语句。

（3）配置 GPIO 方向 GPIODirModeSet()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 GPIODirModeSet()函数，如图 3.9 所示。

```
void GPIODirModeSet(unsigned long ulPort, unsigned char ucPins,
                    unsigned long ulPinIO);
```

功能：配置 GPIO 方向。

```
172 void
173 GPIODirModeSet(unsigned long ulPort, unsigned char ucPins,
174                 unsigned long ulPinIO)
175 {
176     //
177     // Check the arguments.
178     //
179     ASSERT(GPIOBaseValid(ulPort));
180     ASSERT((ulPinIO == GPIO_DIR_MODE_IN) || (ulPinIO == GPIO_DIR_MODE_OUT));
181
182     //
183     // Set the pin direction and mode.
184     //
185     HWREG(ulPort + GPIO_O_GPIO_DIR) = ((ulPinIO & 1) ?
186                                        (HWREG(ulPort + GPIO_O_GPIO_DIR) | ucPins) :
187                                        (HWREG(ulPort + GPIO_O_GPIO_DIR) & ~(ucPins)));
188 }
```

图 3.9　Blinky 项目 GPIODirModeSet()函数

参数说明：

- ulPort：GPIO 端口基地址，在 hw_memmap.h 中定义。
- ucPins：引脚位权，4 个 GPIO 组（GPIOA0～3）中 8 个 GPIO 的引脚位权从低到高依次为 0x01、0x02、0x04、0x08、0x10、0x20、0x40 和 0x80。
- ulPinIO：GPIO 引脚方向，包括下列值。
 - ✧ GPIO_DIR_MODE_IN：输入，0x0000 0000。
 - ✧ GPIO_DIR_MODE_OUT：输出，0x0000 0001。

GPIODirModeSet()的声明在 gpio.h 中，代码在 gpio.c 中。

② 单击调试工具栏中的“Step Return”（单步返回）按钮，运行上述函数，然后退出函数，返回到 PinMuxConfig()函数的 MAP_PinTypeGPIO()语句。

③ 单击调试工具栏中的“Step Return”（单步返回）按钮，对其他引脚进行配置，然后退出函数，返回 main()函数的 GPIO_IF_LedConfigure()语句。

3．LED 操作

LED 操作主要包括下列内容。

- 配置 LED：GPIO_IF_LedConfigure()。
- 熄灭 LED：GPIO_IF_LedOff()（GPIOPinWrite()）。
- 点亮 LED：GPIO_IF_LedOn()（GPIOPinWrite()）。

程序代码在 gpio_if.c 中，其中调用了相应的 GPIO API 函数，主要是 GPIOPinWrite()，和 GPIOPinWrite()对应的是 GPIOPinRead()。

（1）配置 LED 函数 GPIO_IF_LedConfigure()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 GPIO_IF_LedConfigure()。

```
void GPIO_IF_LedConfigure(unsigned char ucPins);
```

功能：配置 LED。

参数说明：ucPins 表示引脚定义。

GPIO_IF_LedConfigure()通过调用 GPIO_IF_GetPortNPin()获得 LED 使用 GPIO 的端口基地址和引脚位权，用于后续的 LED 操作。

② 单击调试工具栏中的“Step Into”（单步进入）按钮三次，进入 GPIO_IF_GetPortNPin()，如图 3.10 所示。

```
337 void
338 GPIO_IF_GetPortNPin(unsigned char ucPin,
339                     unsigned int *puiGPIOPort,
340                         unsigned char *pucGPIOPin)
341 {
342     //
343     // Get the GPIO pin from the external Pin number
344     //
345     *pucGPIOPin = 1 << (ucPin % 8);
346
347     //
348     // Get the GPIO port from the external Pin number
349     //
350     *puiGPIOPort = (ucPin / 8);
351     *puiGPIOPort = ulReg[*puiGPIOPort];
352 }
```

图 3.10　Blinky 项目 GPIO_IF_GetPortNPin()函数

```
void GPIO_IF_GetPortNPin(unsigned char ucPin, unsigned int *puiGPIOPort,
                         unsigned char *pucGPIOPin);
```

功能：根据引脚名称号获取端口基地址和引脚位权。

参数说明：

- ucPin：引脚名称号，GPIO0～39。
- *puiGPIOPort：GPIO 端口基地址指针，端口基地址通过数组 ulReg[]获得，数组下标是引脚名称号 ucPin 整除 8，取值为 0～4，对应 GPIOA0～4。
- *pucGPIOPin：GPIO 引脚位权指针，引脚位权是 1 左移 0～7 位，左移位数是引脚名称号 ucPin 余除 8，取值为 0～7，对应 GPIOAx.0～7。

③ 单击调试工具栏中的“Step Return”（单步返回）按钮两次，获取端口基地址和对应的引脚位权，然后退出函数，返回 main()函数的 GPIO_IF_LedOff()语句。

（2）熄灭 LED 函数 GPIO_IF_LedOff()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 GPIO_IF_LedOff()。

```
void GPIO_IF_LedOff(char ledNum);
```

功能：熄灭 LED。

参数说明：ledNum，LED 标识，MCU_ALL_LED_IND 表示红黄绿 3 个 LED。

GPIO_IF_LedOff()通过调用 GPIO_IF_Set()熄灭 LED。

② 单击调试工具栏中的“Step Into”（单步进入）按钮三次，进入 GPIO_IF_Set()。

```
void GPIO_IF_Set(unsigned char ucPin, unsigned int uiGPIOPort,
                 unsigned char ucGPIOPin, unsigned char ucGPIOValue);
```

功能：根据引脚名称号、端口基地址和引脚位权设置 GPIO 引脚的值。

参数说明：

- ucPin：引脚名称号，GPIO0～31。
- uiGPIOPort：GPIO 端口基地址。

- ucGPIOPin：GPIO 引脚位权。
- ucGPIOValue：GPIO 引脚值，0 或 1 对于 CC3200 LaunchPad，0 表示 LED 熄灭，1 表示 LED 点亮。

GPIO_IF_Set()通过调用 GPIO API 函数 GPIOPinWrite()对 GPIO 引脚进行写操作。

（3）写 GPIO 引脚 GPIOPinWrite()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮三次，进入 GPIOPinWrite()，如图 3.11 所示。

```
613 void
614 GPIOPinWrite(unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)
615 {
616     //
617     // Check the arguments.
618     //
619     ASSERT(GPIOBaseValid(ulPort));
620
621     //
622     // Write the pins.
623     //
624     HWREG(ulPort + (GPIO_O_GPIO_DATA + (ucPins << 2))) = ucVal;
625 }
```

图 3.11　Blinky 项目 GPIOPinWrite()函数

```
void GPIOPinWrite(unsigned long ulPort, unsigned char ucPins,
                  unsigned char ucVal);
```

功能：根据端口基地址和引脚位权写 GPIO 引脚。

参数说明：

- ulPort：GPIO 端口基地址。
- ucPins：GPIO 引脚位权，4 个 GPIO 组（GPIOA0～3）中 8 个 GPIO 的引脚位权从低到高依次为 0x01、0x02、0x04、0x08、0x10、0x20、0x40 和 0x80。
- ucVal：GPIO 引脚值。

GPIOPinWrite()的声明在 gpio.h 中，代码在 gpio.c 中。

和写 GPIO 引脚 GPIOPinWrite()对应的读 GPIO 引脚 GPIOPinRead()的函数格式为

```
long GPIOPinRead(unsigned long ulPort, unsigned char ucPins);
```

功能：根据端口基地址和引脚位权读 GPIO 引脚。

参数说明：

- ulPort：GPIO 端口基地址。
- ucPins：GPIO 引脚位权。

返回值：GPIO 引脚值。

GPIOPinRead()的声明在 gpio.h 中，代码在 gpio.c 中。

② 单击调试工具栏中的“Step Return”（单步返回）按钮，运行写 GPIO 引脚，开发板上的绿 LED 熄灭，然后退出 GPIOPin Write()函数，返回 GPIO_IF_Set()函数。

③ 再单击调试工具栏中的“Step Return”（单步返回）按钮，退出 GPIO_IF_Set()函数，返回 GPIO_IF_LedOff()函数的第二条 GPIO_IF_Set()语句。

④ 单击调试工具栏中的“Step Over”（单步跨越）按钮，运行第二条 GPIO_IF_Set()语句，开发板上的黄 LED 熄灭。

⑤ 再单击调试工具栏中的“Step Over”（单步跨越）按钮，运行第三条 GPIO_IF_Set()语句，开发板上的红 LED 熄灭（默认已熄灭）。

⑥ 单击调试工具栏中的“Step Return”（单步返回）按钮，退出 GPIO_IF_LedOff()函数，返回 main()函数的 LEDBlinkyRoutine()语句。

（4）LED 闪烁 LEDBlinkyRoutine()。LED 闪烁通过延时熄灭 LED 和延时点亮 LED 实现。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 LEDBlinkyRoutine()函数，如图 3.12 所示。

```
120 void LEDBlinkyRoutine()
121 {
122     //
123     // Toggle the lines initially to turn off the LEDs.
124     // The values driven are as required by the LEDs on the LP.
125     //
126     GPIO_IF_LedOff(MCU_ALL_LED_IND);
127     while(1)
128     {
129         //
130         // Alternately toggle hi-low each of the GPIOs
131         // to switch the corresponding LED on/off.
132         //
133         MAP_UtilsDelay(8000000);
134         GPIO_IF_LedOn(MCU_RED_LED_GPIO);
135         MAP_UtilsDelay(8000000);
136         GPIO_IF_LedOff(MCU_RED_LED_GPIO);
137         MAP_UtilsDelay(8000000);
138         GPIO_IF_LedOn(MCU_ORANGE_LED_GPIO);
139         MAP_UtilsDelay(8000000);
140         GPIO_IF_LedOff(MCU_ORANGE_LED_GPIO);
141         MAP_UtilsDelay(8000000);
142         GPIO_IF_LedOn(MCU_GREEN_LED_GPIO);
143         MAP_UtilsDelay(8000000);
144         GPIO_IF_LedOff(MCU_GREEN_LED_GPIO);
145     }
146
147 }
```

图 3.12　Blinky 项目 LEDBlinkyRoutine()函数

② 单击调试工具栏中的“Step Over”（单步跨越）按钮，运行 GPIO_IF_LedOff()和 UtilsDelay()。

③ 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 GPIO_IF_LedOn()。

```
void GPIO_IF_LedOn(char ledNum);
```

功能：点亮 LED。

参数说明：ledNum，LED 标识，包括下列值。

- MCU_RED_LED_GPIO：红 LED。
- MCU_ORANGE_LED_GPIO：黄（橙）LED。
- MCU_GREEN_LED_GPIO：绿 LED。

GPIO_IF_LedOn()也通过调用 GPIO_IF_Set()和 GPIO API GPIOPinWrite()点亮 LED。

注意：GPIO_IF_LedOn()和 GPIO_IF_LedOff()都通过调用 GPIO_IF_Set()设置 LED，但调用 GPIO_IF_Set()的第 4 个参数 ucGPIOValue 的值不同。

④ 单击调试工具栏中的“Step Return”（单步返回）按钮，运行 GPIO_IF_LedOn()语句，开发板上的红 LED 点亮。

⑤ 单击调试工具栏中的“Step Over”（单步跨越）按钮两次，运行 UtilsDelay()和 GPIO_IF_LedOff()语句，开发板上的红 LED 熄灭。

⑥ 单击调试工具栏中的“Step Over”（单步跨越）按钮两次，运行 UtilsDelay()和 GPIO_IF_LedOn()语句，开发板上的黄 LED 点亮。

⑦ 单击调试工具栏中的“Step Over”（单步跨越）按钮两次，运行 UtilsDelay()和 GPIO_IF_LedOff()语句，开发板上的黄 LED 熄灭。

⑧ 单击调试工具栏中的“Resume”（运行）按钮运行程序，开发板上的 3 个 LED 循环闪烁。

⑨ 单击调试工具栏中的“Terminate”（结束）按钮，结束调试返回编辑界面。

综上所述，Blinky 项目使用的 API 函数如表 3.1 所示。

表 3.1 Blinky 项目使用的 API 函数

功 能	函数名	调用函数	代码文件
设置中断向量表	IntVTableBaseSet()	BoardInit()	interrupt.c
允许处理器中断	IntMasterEnable()	BoardInit()	interrupt.c
允许 CPU 中断	CPUcpsie()	IntMasterEnable()	cpu.c
允许中断	IntEnable()	BoardInit()	interrupt.c
初始化 MCU	PRCMCC3200MCUInit()	BoardInit()	prcm.c
允许外设时钟	**PRCMPeripheralClkEnable()**	PinMuxConfig()	prcm.c
配置 GPIO 引脚	**PinTypeGPIO()**	PinMuxConfig()	pin.c
设置引脚类型	PinConfigSet()	PinTypeGPIO()	pin.c
设置引脚模式	PinModeSet()	PinTypeGPIO()	pin.c
配置 GPIO 方向	**GPIODirModeSet()**	PinMuxConfig()	gpio.c
配置 LED	GPIO_IF_LedConfigure()	main()	gpio_if.c
获得端口和引脚	GPIO_IF_GetPortNPin()	GPIO_IF_LedConfigure()	gpio_if.c
熄灭 LED	GPIO_IF_LedOff()	main()、LEDBlinkyRoutine()	gpio_if.c
点亮 LED	GPIO_IF_LedOn()	LEDBlinkyRoutine()	gpio_if.c
设置 GPIO	GPIO_IF_Set()	GPIO_IF_LedOff()、GPIO_IF_LedOn()	gpio_if.c
写 GPIO 引脚	**GPIOPinWrite()**	GPIO_IF_Set()	gpio.c
读 GPIO 引脚	**GPIOPinRead()**	GPIO_IF_Get()	gpio.c

3.1.2 GPIO 使用简介

GPIO 的使用主要包括引脚复用配置和 GPIO 读写操作等。

1. 引脚复用配置

引脚复用配置包括允许 GPIO 时钟 PRCMPeripheralClkEnable()、配置 GPIO 引脚类型 PinTypeGPIO()和配置 GPIO 引脚方向 GPIODirModeSet()，API 声明如下。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);

void PinTypeGPIO(unsigned long ulPin, unsigned long ulPinMode,
                 tBoolean bOpenDrain);

void GPIODirModeSet(unsigned long ulPort, unsigned char ucPins,
                    unsigned long ulPinIO);
```

PinTypeGPIO()的 ulPin 参数与 GPIODirModeSet()的 ulPort 和 ucPins 参数的关系为：首先根据 ulPin（引脚号）从附录 C 中找到对应的引脚名称号，然后根据引脚名称号确定 ulPort（端口基地址）和 ucPins（引脚位权），参见 GPIO_IF_GetPortNPin()内容。

例如，PIN_01 对应的引脚名称是 GPIO10，GPIO10 是第 2 组的第 3 个引脚，所以引脚的端口基地址是 GPIOA1_BASE，引脚位权是 2^{3-1}=4，参见 PinMuxConfig()内容。

GPIODirModeSet()的 ulPinIO 参数包括下列值。

- GPIO_DIR_MODE_IN：输入，0x0000 0000。
- GPIO_DIR_MODE_OUT：输出，0x0000 0001。

2．GPIO 读写操作

GPIO 读写操作分别使用读 GPIO 引脚 GPIOPinRead()和写 GPIO 引脚 GPIOPinWrite()实现，API 声明如下。

```
long GPIOPinRead(unsigned long ulPort, unsigned char ucPins);

void GPIOPinWrite(unsigned long ulPort, unsigned char ucPins,
                  unsigned char ucVal);
```

3.1.3　按键控制 LED 闪烁时间程序设计

在 GPIO 示例程序的基础上稍加修改，可以用按键控制 LED 的闪烁时间，同时直接使用 GPIO API 对 GPIO 进行操作，具体步骤如下。

1．修改 PinMuxConfig()函数

在 pinmux.c 文件的 PinMuxConfig()函数中添加下列语句，实现按键接口的配置。

```
//允许 GPIOA2 时钟，连接 SW2 的 GPIO_22 在 GPIOA2（GPIO_16～GPIO_23）
MAP_PRCMPeripheralClkEnable(PRCM_GPIOA2, PRCM_RUN_MODE_CLK);

//配置 PIN_04(GPIO_13)为 GPIO 输入
MAP_PinTypeGPIO(PIN_04, PIN_MODE_0, false);
MAP_GPIODirModeSet(GPIOA1_BASE, 0x20, GPIO_DIR_MODE_IN);

//配置 PIN_15(GPIO_22)为 GPIO 输入
MAP_PinTypeGPIO(PIN_15, PIN_MODE_0, false);
MAP_GPIODirModeSet(GPIOA2_BASE, 0x40, GPIO_DIR_MODE_IN);
```

2．修改 main()函数

修改 main()函数的内容如下。

```
unsigned long ulCount = 8000000;                    //定义延时初值
BoardInit();                                        //初始化开发板
PinMuxConfig();                                     //引脚复用配置
MAP_GPIOPinWrite(GPIOA1_BASE, 0xe, 0);              //3 个 LED 熄灭

while(1)
{
    if(MAP_GPIOPinRead(GPIOA1_BASE, 0x20))          //按下 SW3
    {
        if(ulCount > 1000000)
            ulCount -= 1000000;                     //减延时值
    }

    if(MAP_GPIOPinRead(GPIOA2_BASE, 0x40))          //按下 SW2
    {
        if(ulCount < 8000000)
            ulCount += 1000000;                     //加延时值
    }

    MAP_UtilsDelay(ulCount);
    MAP_GPIOPinWrite(GPIOA1_BASE, 2, 2);            //红 LED 点亮
    MAP_UtilsDelay(ulCount);
    MAP_GPIOPinWrite(GPIOA1_BASE, 2, 0);            //红 LED 熄灭
    MAP_UtilsDelay(ulCount);
    MAP_GPIOPinWrite(GPIOA1_BASE, 4, 4);            //黄 LED 点亮
    MAP_UtilsDelay(ulCount);
    MAP_GPIOPinWrite(GPIOA1_BASE, 4, 0);            //黄 LED 熄灭
    MAP_UtilsDelay(ulCount);
    MAP_GPIOPinWrite(GPIOA1_BASE, 8, 8);            //绿 LED 点亮
    MAP_UtilsDelay(ulCount);
    MAP_GPIOPinWrite(GPIOA1_BASE, 8, 0);            //绿 LED 熄灭
}
```

运行程序，3 个 LED 延时闪烁。按下 SW3，3 个 LED 闪烁延时时间逐遍缩短，闪烁 8 次后，延时时间最短。按下 SW2，3 个 LED 闪烁延时时间逐遍加长，闪烁 8 次后，延时时间恢复到最长。因为主程序采用查询方式检测按键，3 个 LED 闪烁一遍后按键才能再起作用，所以按键反应比较迟钝。为了及时响应按键操作，可以采用中断方式检测按键。

3.1.4 PinMux 使用介绍

如果开发板上的用户按键和用户 LED 使用的 GPIO 引脚与 CC3200 LaunchPad 不同，例如，两个用户按键使用 GPIO7 和 GPIO8，3 个用户 LED 分别使用 GPIO22、GPIO28 和 GPIO30，可以像 3.1.3 节中”修改 PinMuxConfig()函数”一样直接修改 PinMuxConfig()函数的内容，但 TI 强烈建议使用 CC3200 引脚复用实用工具 PinMux 获得需要的引脚配置。

1．用 PinMux 生成引脚复用配置文件

（1）打开 TI Pin Mux Tool，选择 CC3200 器件，单击“Start”按钮，如图 3.13 所示。



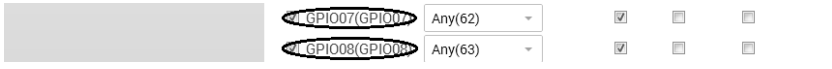
图 3.13　Pin Mux 选择器件

（2）单击“Available Peripherals”（可用外设）下“GPIO”右边的加号“+”添加“My GPIO 1”，单击“My GPIOs”下的“GPIO Signals”（GPIO 信号），消除所有信号引脚前的对号（不选择所有信号），如图 3.14 所示。



图 3.14　Pin Mux 可用外设

（3）单击“GPIO07”和“GPIO08”，选择这两个输入引脚；单击“GPIO22”、“GPIO28”和“GPIO30”，选择这三个输出引脚，并单击信号右边的“Output”（输出）项，如图 3.15 所示。



(a) 输入引脚



(b) 输出引脚

图 3.15　Pin Mux 输入引脚和输出引脚

（4）单击“Generate”（生成）按钮，打开“浏览文件夹”对话框，浏览到“D:\TI\ CC3200 SDK_1.1.0\example\blinky”文件夹，如图 3.16 所示。

（5）单击“确定”按钮，在“blinky”文件夹生成下列 4 个文件。

- pin_mux_config.c：引脚复用配置源文件。
- pin_mux_config.h：引脚复用配置头文件。
- rom_pin_mux_config.c：ROM 引脚复用配置源文件。
- summary.csv：引脚复用配置表。

（6）关闭 Pin Mux，单击“Yes”（是）。

2．替换引脚复用配置文件

（1）在 CCS 的“Project Explorer”视图中右击 blinky 项目下的“pinmux.c”，在弹出菜单中单击“Delete”（删除）将 pinmux.c 从项目中删除，在弹出对话框中单击“OK”按钮确认删除。

（2）在 CCS 的“Project Explorer”视图中右击“blinky [Active – Release]”，在弹出菜单中单击“Add Files…”（添加文件），选择 rom_pin_mux_config.c，在弹出对话框“File Operation”（文件操作）中选择“Copy files”（复制文件），单击“OK”按钮，将 rom_pin_ mux_config.c 添加到项目中，如图 3.17 所示。



图 3.16　Pin Mux“浏览文件夹”对话框

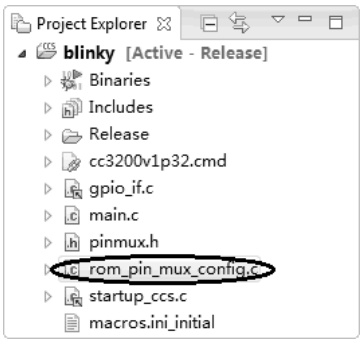


图 3.17　Blinky 项目替换引脚复用配置文件

（3）重复步骤（1）和步骤（2），将 pinmux.h 替换为 pin_mux_config.h。

（4）将 main.c 中包含的头文件名 pinmux.h 修改为 pin_mux_config.h。

3.2　UART 接口应用

串行接口分为异步串行接口和同步串行接口两种。异步串行接口统称为通用异步收发器（UART）接口，同步串行接口有 SPI 和 I2C 等，同步串行接口除了包含数据线（SPI 有两根单向数据线 MISO 和 MOSI，I2C 有一根双向数据线 SDA）外，还包含时钟线（SPI 和 I2C 的时钟线分别是 SCK 和 SCL）。

SPI 和 I2C 都可以连接多个从设备，但两者选择从设备的方法不同：SPI 通过硬件（NSS 引脚）实现，而 I2C 通过软件（地址）实现。

为了使不同电压输出的器件能够互连，I2C 的数据线 SDA 和时钟线 SCL 开漏输出，使用时必须通过电阻连接到电源线。

同步串行接口可以用专用接口电路硬件实现，也可以用通用并行接口软件实现。

3.2.1　UART 简介

UART 的相关标准规定了接口的机械特性、电气特性和功能特性等，UART 的电气特性标准包括 RS-232C、RS-422、RS-423 和 RS-485 等，其中 RS-232C 是最常用串行通信标准。

RS-232C 是数据终端设备（DTE）和数据通信设备（DCE）之间串行二进制数据交换接口技术标准，其中 DTE 包括微机、微控制器和打印机等，DCE 包括调制解调器 MODEM、GSM 模块和 Wi-Fi 模块等。

RS-232C 机械特性规定 RS-232C 使用 25 针 D 型连接器，后来简化为 9 针 D 型连接器。

RS-232C 电气特性采用负逻辑：逻辑“1”的电平低于–3 V，逻辑“0”的电平高于+3 V，这和 TTL 的正逻辑（逻辑“1”为高电平，逻辑“0”为低电平）不同，因此通过 RS-232C 和 TTL 器件通信时必须进行电平转换。

目前微控制器的 UART 接口采用的是 TTL 正逻辑，和 TTL 器件连接不需要电平转换。和采用负逻辑的计算机相连时需要进行电平转换，或使用 UART-USB 转换器连接。

RS-232C 功能特性规定各引脚的功能，9 针 D 型连接器的引脚功能如表 3.2 所示，其中最常用的引脚只有 3 个：RXD（接收数据）、TXD（发送数据）和 GND（地）。

表 3.2　RS-232C 引脚功能

引脚	名称	功能	DTE 方向	DCE 方向	引脚	名称	功能	DTE 方向	DCE 方向
1	DCD	载波检测	输入	输出	6	DSR	数据设备准备好	输入	输出
2	RXD	接收数据	输入	输出	7	RTS	请求发送	输出	输入
3	TXD	发送数据	输出	输入	8	CTS	清除发送	输入	输出
4	DTR	数据终端准备好	输出	输入	9	RI	振铃指示	输入	输出
5	GND	地							

从表 3.2 中可以看出，用 RS-232C 连接 DTE 和 DCE 时各引脚直接连接，DTE 的 TXD（输出）直接连接 DCE 的 TXD（输入），DTE 的 RXD（输入）直接连接 DCE 的 RXD（输出）。

如果用 RS-232C 连接 DTE 和 DTE（如微机和微控制器），相关引脚必须交叉连接，DTE1 的 TXD（输出）连接 DTE2 的 RXD（输入），DTE1 的 RXD（输入）连接 DTE2 的 TXD（输出）。

RS-232C 的主要指标有 2 个：数据速率和数据格式。数据速率用波特率表示，数据格式包括 1 个起始位、5～8 个数据位、0～1 个校验位和 1～2 个停止位，如图 3.18 所示。

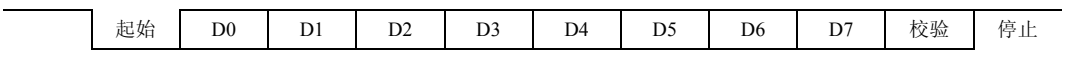


图 3.18　RS-232C 数据格式

通信双方的数据速率和数据格式必须一致，否则无法实现通信。

CC3200 包含两个可编程 UART 接口（UARTA0～1，具体引脚分配参见表 D.2），CC3200 UART 具有下列特性。

- 可编程波特率发生器，允许速度高达 3 Mbps。
- 独立的 16×8 发送（TX）和接收（RX）FIFO，减轻 CPU 中断处理负载。
- 可编程 FIFO 长度，包括提供传统双缓冲接口的单字节操作。
- FIFO 触发阈值包括 1/8、1/4、1/2、3/4 和 7/8。
- 标准的异步通信起始、停止和奇偶校验位。
- 可编程串行接口特性。
 - ✧ 5、6、7 或 8 位数据。
 - ✧ 偶、奇或无校验生成/检测。
 - ✧ 1 或 2 个停止位生成。
- 支持 RTS 和 CTS 调制解调器握手。
- 标准的 FIFO 阈值中断和传输结束中断。
- 使用 DMA 可实现高效传输。
 - ✧ 单独的发送和接收通道。
 - ✧ FIFO 中有数据的单个请求接收和可编程 FIFO 阈值的突发请求接收。
 - ✧ FIFO 中有空间的单个请求发送和可编程 FIFO 阈值的突发请求发送。

3.2.2 UART 示例程序分析

UART 示例程序在 example\uart_demo 目录中，Uart_demo 项目 CC3200 LaunchPad 电路如图 3.19 所示。

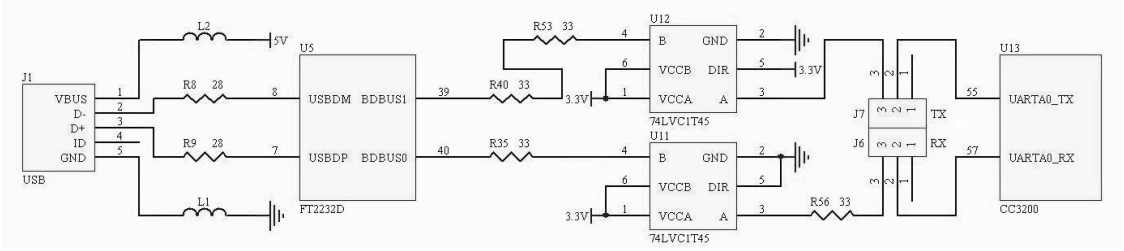


图 3.19　Uart_demo 项目 CC3200 LaunchPad 电路图

CC3200 UARTA0（TX-PIN_55/ RX-PIN_57）经过电平转换电路 74LVC1T45（可以省略）和 UART-USB 转换电路 FT2232D 转换后，通过 USB 接口与 PC 相连。

（1）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（2）在 PC 上打开超级终端 Tera Term。

（3）参考 2.1.2 节中的“CCS 使用”，将 uart_demo 项目导入 CCS，在“Project Explorer”（项目浏览器）视图中可以看出 uart_demo 项目包含下列 4 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。

- uart_if.c：UART 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Uart_demo 项目程序流程图如图 3.20 所示。

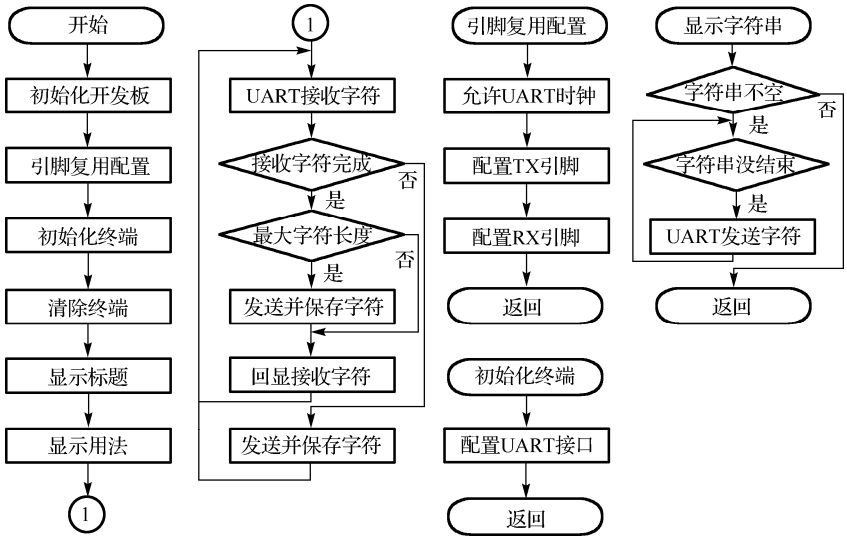


图 3.20　Uart_demo 项目程序流程图

（4）在 CCS 中右击“Project Explorer”（项目浏览器）中的 uart_demo 项目，在弹出菜单中单击“Properties”（属性），打开“属性”对话框，确认“Device”（器件）选择 CC3200，选择“Compiler version”（编译器版本）为 TI v5.2.2，如图 2.17 所示。

（5）在 CCS 中单击“Run”（运行）→“Debug”（调试）或按“F11”键，CCS 首先生成目标文件 uart_demo.out，然后将目标文件 uart_demo.out 下载到 CC3200 并进入调试界面。

（6）单击“Run”（运行）→“Resume”（运行）或单击调试工具栏中的“Resume”（运行）按钮或按“F8”键，运行程序，在超级终端窗口将出现图 3.21 所示的程序标题和用法信息及命令提示符。

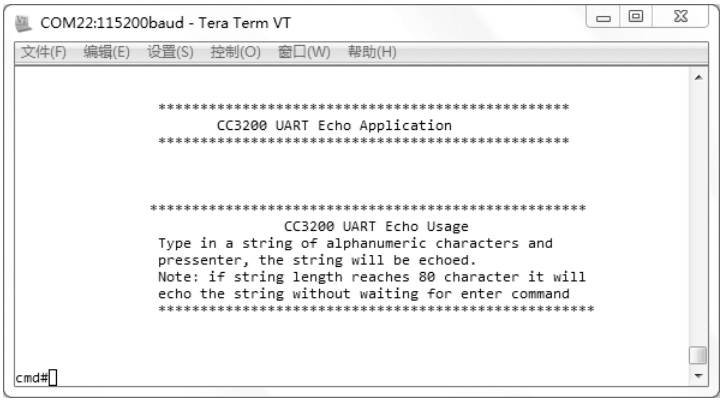


图 3.21　Uart_demo 项目超级终端信息

（7）在超级终端中输入字母数字字符，输入的内容回显在超级终端中，按 Enter（回车）

键，将回显字符串。如果字符串长度达到 80 个字符，将自动回显字符串，不等待按 Enter（回车）键。

（8）单击调试工具栏中的“Suspend”（暂停）按钮暂停程序的运行。

（9）单击调试工具栏中的“Restart”（重新开始）按钮，程序停在 main()函数开始处。

注意：在程序的分析过程中，可随时单击“Restart”（重新开始）按钮复位程序。

下面详细介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- 终端操作。

1．初始化开发板 BoardInit()

初始化开发板包括设置中断向量表、允许处理器中断、允许 SysTick 中断和初始化 MCU 等步骤，调用了相应的中断 API 函数，具体内容和 Blinky 项目相同。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序由 PinMux 应用程序根据设计需要自动生成，程序代码在 pinmux.c 中，其中调用了相应的 PRCM 和 PIN API 函数。

右击 main()中 PinMuxConfig()语句，在弹出菜单中单击“Run to Line”（允许到行），运行到 PinMuxConfig()语句。

单击调试工具栏中的“Step Into”（单步进入）按钮，进入 PinMuxConfig()函数，如图 3.22 所示。

```
55 void
56 PinMuxConfig(void)
57 {
58     //
59     // Enable Peripheral Clocks
60     //
61     MAP_PRCMPeripheralClkEnable(PRCM_UARTA0, PRCM_RUN_MODE_CLK);
62
63     //
64     // Configure PIN_55 for UART0 UART0_TX
65     //
66     MAP_PinTypeUART(PIN_55, PIN_MODE_3);
67
68     //
69     // Configure PIN_57 for UART0 UART0_RX
70     //
71     MAP_PinTypeUART(PIN_57, PIN_MODE_3);
72 }
```

图 3.22　Uart_demo 项目 PinMuxConfig()函数

UART 引脚复用配置包括下列内容。

- 允许 UART 时钟：PRCMPeripheralClkEnable()。
- 配置 UART 引脚：PinTypeUART()。

（1）允许 UART 时钟 PRCMPeripheralClkEnable()。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：外设号，UARTA0 的外设号是 0x0C，在 prcm.h 中定义。
- ulClkFlags：时钟标志，设置值是 PRCM_RUN_MODE_CLK（运行模式时钟）。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

注意：GPIO 和 UART 等外设接口和外设都使用 PRCMPeripheralClkEnable()函数允许外设时钟，不同之处是参数 ulPeripheral（外设号）不同。

单击调试工具栏中的“Step Over”（单步跨越）按钮两次，运行 PRCMPeripheralClkEnable()语句。

（2）配置 UART 引脚 PinTypeUART()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 PinTypeUART()函数，如图 3.23 所示。

```
382 void PinTypeUART(unsigned long ulPin,unsigned long ulPinMode)
383 {
384     //
385     // Set the pin to specified mode
386     //
387     PinModeSet(ulPin,ulPinMode);
388
389     //
390     // Set the pin for standard operation
391     //
392     PinConfigSet(ulPin,PIN_STRENGTH_2MA,PIN_TYPE_STD);
393 }
```

图 3.23　Uart_demo 项目 PinTypeUART()函数

```
void PinTypeUART(unsigned long ulPin, unsigned long ulPinMode);
```

功能：配置 UART 引脚。

参数说明：

- ulPin：引脚号，UARTA0_TX 和 UARTA0_RX 的引脚号分别是 PIN_55 和 PIN_57。
- ulPinMode：引脚模式，PIN_55 和 PIN_57 的 PIN_MODE_3 为 UART 模式。

PinTypeUART()的声明在 pin.h 中，代码在 pin.c 中。PinTypeUART()通过调用 PinModeSet()和 PinConfigSet()实现 UART 引脚的配置。

注意：配置 GPIO 引脚 PinTypeGPIO()和配置 UART 引脚 PinTypeUART()都通过调用 PinModeSet()和 PinConfigSet()实现，两者的不同之处是 PinModeSet()的参数 ulPinMode（引脚模式）不同，前者为 GPIO 模式，后者为 UART 模式。

② 单击调试工具栏中的“Step Return”（单步返回）按钮，运行上述函数，然后退出函数，返回 PinMuxConfig()函数的第二条 PinTypeUART()语句。

③ 单击调试工具栏中的“Step Return”（单步返回）按钮，对 PIN_57 进行配置，然后退出函数，返回 main()函数的 InitTerm()语句。

3．终端操作

终端操作主要包括下列内容。

- 初始化终端：InitTerm()（UARTConfigSetExpClk()）。
- 清除终端：ClearTerm()（UARTCharPut()）。
- 显示标题：DisplayBanner()。
- 显示用法（显示字符串）：Message()。

主要程序代码在 uart_if.c 中，其中调用了相应的 UART API 函数，主要是 UARTConfig SetExpClk()、UARTCharPut()和 UARTCharGet()。

（1）初始化终端 InitTerm()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 InitTerm()函数，如图 3.24 所示。

```
83 void
84 InitTerm()
85 {
86 #ifndef NOTERM
87   MAP_UARTConfigSetExpClk(CONSOLE,MAP_PRCMPeripheralClockGet(CONSOLE_PERIPH),
88                           UART_BAUD_RATE, (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
89                            UART_CONFIG_PAR_NONE));
90 #endif
91   __Errorlog = 0;
92 }
```

图 3.24　Uart_demo 项目 InitTerm()函数

初始化终端 InitTerm()通过调用 UARTConfigSetExpClk()实现，UARTConfigSetExpClk()又通过调用 PRCMPeripheralClkGet()获得 UART 时钟。

② 单击调试工具栏中的“Step Into”（单步进入）按钮两次，进入 PRCMPeripheral ClkGet()。

```
unsigned long PRCMPeripheralClkGet(unsigned long ulPeripheral);
```

功能：获取外设时钟。

参数说明：ulPeripheral，外设号，UARTA0 的外设号是 0x0C，在 prcm.h 中定义。

返回值：外设时钟，UARTA0 的外设时钟是系统时钟 SYS_CLK（80 MHz）。

PRCMPeripheralClkGet()的声明在 prcm.h 中，代码在 prcm.c 中。

③ 单击调试工具栏中的“Step Return”（单步返回）按钮，运行上述函数，然后退出函数，返回 InitTerm()函数的 UARTConfigSetExpClk()语句。

④ 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 UARTConfigSetExpClk()函数，如图 3.25 所示。

```
void UARTConfigSetExpClk(unsigned long ulBase, unsigned long ulUARTClk,
                         unsigned long ulBaud, unsigned long ulConfig);
```

功能：配置 UART 接口。

参数说明：

```
314 void
315 UARTConfigSetExpClk(unsigned long ulBase, unsigned long ulUARTClk,
316                     unsigned long ulBaud, unsigned long ulConfig)
317 {
318     unsigned long ulDiv;
319
320     //
321     // Check the arguments.
322     //
323     ASSERT(UARTBaseValid(ulBase));
324     ASSERT(ulBaud != 0);
325
326     //
327     // Stop the UART.
328     //
329     UARTDisable(ulBase);
```

图 3.25　Uart_demo 项目 UARTConfigSetExpClk()函数（部分）

- ulBase：UART 端口基地址，UARTA0 的端口基地址为 0x4000 C000，在 hw_memmap.h 中定义。
- ulUARTClk：UART 时钟（80 MHz），通过调用 PRCMPeripheralClkGet()获得。
- ulBaud：UART 波特率，设置值为 115 200。
- ulConfig：UART 参数，包括数据位数（5～8 位）、停止位数（1～2 位）和校验位（偶校验、奇校验和无校验）等下列值。
 - ✧ UART_CONFIG_WLEN_8：8 位数据，0x0000 0060。
 - ✧ UART_CONFIG_WLEN_7：7 位数据，0x0000 0040。
 - ✧ UART_CONFIG_WLEN_6：6 位数据，0x0000 0020。
 - ✧ UART_CONFIG_WLEN_5：5 位数据，0x0000 0000。
 - ✧ UART_CONFIG_STOP_TWO：2 个停止位，0x0000 0008。
 - ✧ UART_CONFIG_STOP_ONE：1 个停止位，0x0000 0000。
 - ✧ UART_CONFIG_PAR_EVEN：偶校验，0x0000 0006。
 - ✧ UART_CONFIG_PAR_ODD：奇校验，0x0000 0002。
 - ✧ UART_CONFIG_PAR_NONE：无校验，0x0000 0000。

设置值是 0x60：UART_CONFIG_WLEN_8（8 位数据）、UART_CONFIG_STOP_ONE （1 个停止位）和 UART_CONFIG_PAR_NONE（无校验），参见图 3.24。

UARTConfigSetExpClk()的声明在 uart.h 中，代码在 uart.c 中。

在通过 UARTConfigSetExpClk()配置 UART 接口前，需要禁止 UART，配置 UART 接口后，再允许 UART。

```
void UARTDisable(unsigned long ulBase);
```

功能：禁止 UART。

参数说明：ulBase，UART 端口基地址，UARTA0 的端口基地址为 0x4000 C000。

UARTDisable()的声明在 uart.h 中，代码在 uart.c 中。

```
void UARTEnable(unsigned long ulBase);
```

功能：允许 UART。

参数说明：ulBase，UART 端口基地址，UARTA0 的端口基地址为 0x4000 C000。

UARTEnable()的声明在 uart.h 中，代码在 uart.c 中。

⑤ 单击调试工具栏中的“Step Return”（单步返回）按钮，运行 UARTConfigSetExpClk()函数，然后退出函数，返回 InitTerm()函数。

⑥ 单击调试工具栏中的“Step Return”（单步返回）按钮，退出 InitTerm()函数，返回 main()函数的 ClearTerm()语句。

（2）清除终端 ClearTerm()。

清除终端通过调用 Message("\33[2J\r")实现，Message()又循环调用 UARTCharPut()显示字符串。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 ClearTerm()函数。

② 再单击调试工具栏中的“Step Into”（单步进入）按钮两次，进入 Message()函数，如图 3.26 所示。

```
106 void
107 Message(const char *str)
108 {
109 #ifndef NOTERM
110     if(str != NULL)
111     {
112         while(*str!='\0')
113         {
114             MAP_UARTCharPut(CONSOLE,*str++);
115         }
116     }
117 #endif
118 }
```

图 3.26　Uart_demo 项目 Message()函数

③ 单击调试工具栏中的“Step Into”（单步进入）按钮4 次，进入 UARTCharPut()函数，如图 3.27 所示。

```
1054 void
1055 UARTCharPut(unsigned long ulBase, unsigned char ucData)
1056 {
1057     //
1058     // Check the arguments.
1059     //
1060     ASSERT(UARTBaseValid(ulBase));
1061
1062     //
1063     // Wait until space is available.
1064     //
1065     while(HWREG(ulBase + UART_O_FR) & UART_FR_TXFF)
1066     {
1067     }
1068
1069     //
1070     // Send the char.
1071     //
1072     HWREG(ulBase + UART_O_DR) = ucData;
1073 }
```

图 3.27　Uart_demo 项目 UARTCharPut()函数

```
void UARTCharPut(unsigned long ulBase, unsigned char ucData);
```

功能：UART 发送字符。

参数说明：

- ulBase：UART 端口基地址，UARTA0 的端口基地址为 0x4000 C000。
- ucData：发送数据。

UARTCharPut()的声明在 uart.h 中，代码在 uart.c 中。

④ 单击调试工具栏中的“Step Return”（单步返回）按钮3 次，运行 UARTCharPut()函数，然后依次退出 UARTCharPut()、Message()和 ClearTerm()函数，返回 main()函数的 DisplayBanner()语句，超级终端中的内容被清除。

（3）显示标题 DisplayBanner()。

单击调试工具栏中的“Step Over”（单步跨越）按钮，运行显示标题 DisplayBanner()函数，DisplayBanner()通过调用 Report()在超级终端中显示下列标题。

```
*******************************************************************************
                          CC3200 UART Echo Application
*******************************************************************************
```

Report()是格式字符串显示函数，通过调用系统函数 vsnprintf()实现格式字符串的转换，再调用 Message()实现格式字符串的显示，功能和 printf()类似。

（4）显示用法 Message()。

右击 while(1)中的“cCharacter = UartGetChar()”语句，在弹出菜单中单击“Run to Line”（运行到行），运行到“cCharacter = UartGetChar()”语句，超级终端中显示下列程序用法信息和命令提示符“cmd#”。

```
*******************************************************************************
                                  CC3200 UART Echo Usage
Type in a string of alphanumeric characters
and press enter, the string will be echoed.
Note: if string length reaches 80 character it will
echo the string without waiting for enter command
*******************************************************************************
```

（5）UART 接收字符 UARTCharGet()。

① 单击调试工具栏中的“Step Into”（单步进入）按钮，进入 UARTCharGet()函数，如图 3.28 所示。

```
970 long
971 UARTCharGet(unsigned long ulBase)
972 {
973     //
974     // Check the arguments.
975     //
976     ASSERT(UARTBaseValid(ulBase));
977
978     //
979     // Wait until a char is available.
980     //
981     while(HWREG(ulBase + UART_O_FR) & UART_FR_RXFE)
982     {
983     }
984
985     //
986     // Now get the char.
987     //
988     return(HWREG(ulBase + UART_O_DR));
989 }
```

图 3.28　Uart_demo 项目 UARTCharGet()函数

```
long UARTCharGet(unsigned long ulBase);
```

功能：UART 接收字符。

参数说明：ulBase，UART 端口基地址，UARTA0 的端口基地址为 0x4000 C000。

返回值：接收数据。

UARTCharGet()的声明在 uart.h 中，代码在 uart.c 中。

注意：main.c 中使用了下列宏定义。

```
#define CONSOLE         UARTA0_BASE
#define UartGetChar()   MAP_UARTCharGet(CONSOLE)
#define UartPutChar(c) MAP_UARTCharPut(CONSOLE, c)
```

② 单击调试工具栏中的“Step Return”（单步返回）按钮，运行 UARTCharGet()函数，在超级终端中输入任一字符（如“1”），退出 UARTCharGet()函数，返回到 main()函数中的“cCharacter = UartGetChar()”语句。

③ 单击调试工具栏中的“Step Over”（单步跨越）按钮，运行赋值语句 cCharacter = UartGetChar()。

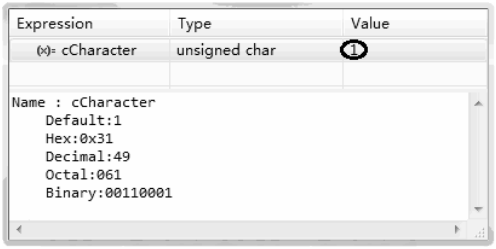


图 3.29 表达式窗口

④ 将鼠标移到 main.c 中的“cCharacter”处，弹出“Expression”（表达式）窗口，如图 3.29 所示。

⑤ 单击“cCharacter”右边“Value”下边的数值“1”，输入“2”、“0x32”或“50”，然后按“Enter”（回车）键，cCharacter 的值变为 2。

（6）回显接收字符 UARTCharPut()。

单个字符的回显通过调用 UartPutChar()即 UARTCharPut()实现，按 Enter（回车）键或字符串长度达到 80 个字符时通过调用 Report()实现字符串回显。

① 单击调试工具栏中的“Step Over”（单步跨越）按钮3 次，运行 UartPutChar()语句发送字符，在超级终端的命令提示符后显示“2”。

② 双击“cCharacter = UartGetChar()”语句行号左侧的断点标志清除断点。

③ 在“Report("\n\rcmd#%s\n\rcmd#", cString)”语句处设置断点。

④ 单击调试工具栏中的“Resume”（运行）按钮，运行程序。

⑤ 在超级终端中输入字母数字字符，输入的字符回显在超级终端中。

⑥ 在超级终端中按“Enter”（回车键）或输入字符达到 80 个时，程序停在断点处。

⑦ 单击调试工具栏中的“Step Over”（单步跨越）按钮，运行 Report()语句发送字符串，输入的字符串回显在超级终端中。

⑧ 双击“Report("\n\rcmd#%s\n\rcmd#", cString)”语句行号左侧的断点标志清除断点。

⑨ 单击调试工具栏中的“Terminate”（结束）按钮，结束调试返回编辑界面。

综上所述，Uart_demo 项目使用的与 UART 有关的 API 函数如表 3.3 所示。

表 3.3　Uart_demo 项目使用的与 UART 有关的 API 函数

功　能	函数名	调用函数	代码文件
允许外设时钟	**PRCMPeripheralClkEnable()**	PinMuxConfig()	Prcm.c
配置 UART 引脚	**PinTypeUART()**	PinMuxConfig()	pin.c
设置引脚模式	PinModeSet()	PinTypeUART()	pin.c
设置引脚类型	PinConfigSet()	PinTypeUART()	pin.c
初始化终端	InitTerm()	main()	uart_if.c
配置 UART 接口	**UARTConfigSetExpClk()**	InitTerm()	uart.c
获得外设时钟	PRCMPeripheralClkGet()	InitTerm()	Prcm.c
禁止 UART	UARTDisable()	UARTConfigSetExpClk()	uart.c
允许 UART	UARTEnable()	UARTConfigSetExpClk()	uart.c
清除终端	ClearTerm()	main()	uart_if.c
显示字符串	Message()	ClearTerm()、Report()、main()	uart_if.c
显示格式字符串	Report()	DisplayBanner()、main()	uart_if.c
UART 发送字符	**UARTCharPut()**	Message()、main()	uart.c
UART 接收字符	**UARTCharGet()**	main()	uart.c

3.2.3　UART 使用简介

UART 的使用主要包括引脚复用配置、初始化终端和 UART 数据操作等。

1．引脚复用配置

引脚复用配置包括允许 UART 时钟 PRCMPeripheralClkEnable()和配置 UART 引脚 PinTypeUART()，API 声明如下。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);

void PinTypeUART(unsigned long ulPin, unsigned long ulPinMode);
```

2．初始化终端

初始化终端主要是配置 UART 接口 UARTConfigSetExpClk()，用于配置 UART 波特率和数据格式（包括数据位数、停止位数和校验位等），其中包含允许 UART UARTEnable()和禁止 UART UARTDisable()，API 声明如下。

```
void UARTConfigSetExpClk(unsigned long ulBase, unsigned long ulUARTClk,
                         unsigned long ulBaud, unsigned long ulConfig);

void UARTEnable(unsigned long ulBase);

void UARTDisable(unsigned long ulBase);
```

UARTConfigSetExpClk()的 ulConfig 参数包括数据位数（5～8 位）、停止位数（1～2 位）和校验位（偶校验、奇校验和无校验）等下列值。

- UART_CONFIG_WLEN_8：8 位数据，0x0000 0060。
- UART_CONFIG_WLEN_7：7 位数据，0x0000 0040。
- UART_CONFIG_WLEN_6：6 位数据，0x0000 0020。
- UART_CONFIG_WLEN_5：5 位数据，0x0000 0000。
- UART_CONFIG_STOP_TWO：2 个停止位，0x0000 0008。
- UART_CONFIG_STOP_ONE：1 个停止位，0x0000 0000。
- UART_CONFIG_PAR_EVEN：偶校验，0x0000 0006。
- UART_CONFIG_PAR_ODD：奇校验，0x0000 0002。
- UART_CONFIG_PAR_NONE：无校验，0x0000 0000。

3．UART 数据操作

UART 数据操作分别使用 UART 发送字符 UARTCharPut()和 UART 接收字符 UART CharGet()实现，API 声明如下。

```
void UARTCharPut(unsigned long ulBase, unsigned char ucData);

long UARTCharGet(unsigned long ulBase);
```

3.2.4 UART 示例程序小改进

UART 示例程序主程序的 while(1)循环中“发送并保存字符”处理出现了两次，可以将其合并，同时可将接收和发送字符宏定义 UartGetChar()和 UartPutChar()改用 UART API 函数 UARTCharGet()和 UARTCharPut()，改进后的程序如下。

```
while(1)
{
    cCharacter = MAP_UARTCharGet(UARTA0_BASE);      //接收字符
    MAP_UARTCharPut(UARTA0_BASE, cCharacter);       //发送字符
    cString[iStringLength] = cCharacter;            //保存字符
    iStringLength++;                                //修改长度
    //接收字符完成
    if(cCharacter == '\r' || (iStringLength >= MAX_STRING_LENGTH-1))
    {
        cString[iStringLength] = '\0';              //添加字符串结束
        iStringLength = 0;
        Report("\n\rcmd#%s\n\rcmd#", cString);      //回显接收字符
    }
}
```

3.3 SPI 接口应用

串行外设接口 SPI 是工业标准串行协议，通常用于嵌入式系统，将微处理器连接到各种片外传感器、转换器、存储器和控制设备。

SPI 可以实现主设备或从设备协议，当配置为主设备时，SPI 可以连接多达 16 个独立的从设备，发送数据和接收数据寄存器的宽度可配置为 8 位或 16 位。

SPI 使用 2 根数据线、1 根时钟线和 1 根控制线实现串行通信。

- 主出从入 MOSI：主设备输出数据，从设备输入数据。
- 主入从出 MISO：主设备输入数据，从设备输出数据。
- 串行时钟 SCK：主设备输出，从设备输入，用于同步数据位。
- 从设备选择 NSS：主设备输出，从设备输入，低电平有效。

SPI 时钟极性和时钟相位所有组合的信号波形如图 3.30 所示。

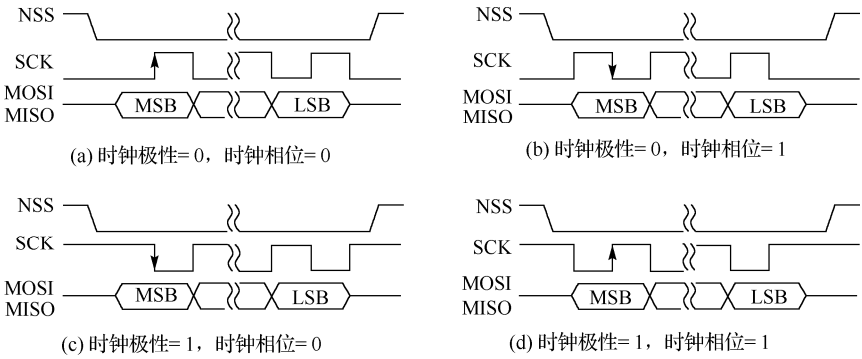


图 3.30　SPI 时钟极性和时钟相位所有组合的信号波形

SCK 时钟极性为 0 时初始电平是低电平，为 1 时初始电平是高电平。时钟相位为 0 时第一个边沿采样数据，为 1 时第二个边沿采样数据。

CC3200 包含 3 个 SPI 接口，即 SSPI、GSPI 和 LSPI，SSPI 用于连接片外 Flash，GSPI 是通用 SPI，LSPI 用于连接片内 Wi-Fi。

CC3200 GSPI 具有下列特性。

- 可编程工作在主模式和从模式。
- 3 线和 4 线模式。
- 全双工和半双工。
- 串行时钟频率、极性和相位可编程。
- 工作频率高达 20 MHz。
- 片选极性可编程。
- 第一个发送字前的延迟可编程。
- 片选和外部时钟之间的定时控制可编程。
- 从模式下两个连续字间没有死循环。
- 8、16、32 位字长。
- 使用 DMA 可实现高效传输。

3.3.1　SPI 示例程序分析

SPI 示例程序在“example\spi_demo”目录中，Spi_demo 项目 CC3200 LaunchPad 电路如图 3.31 所示。

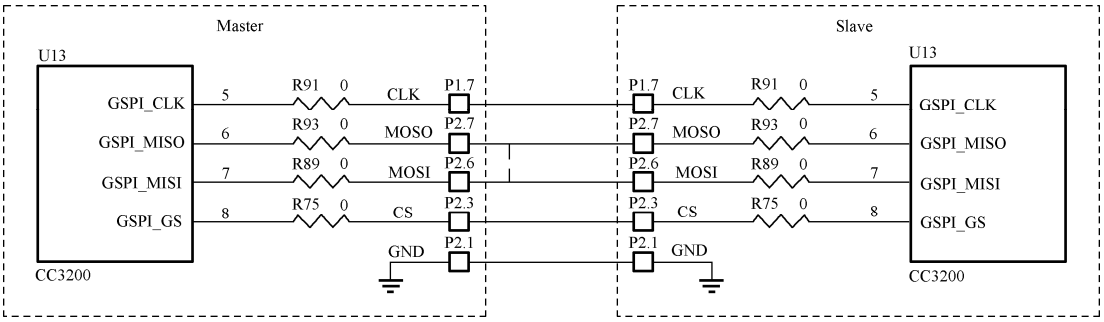


图 3.31　Spi_demo 项目 CC3200 LaunchPad 电路图

两个 CC3200 LaunchPad 通过 GSPI 接口进行通信，一个作为主设备（Master），一个作为从设备（Slave）。也可以用短路块连接 P2.6（MOSI）和 P2.7（MISO），实现自环回，下面以自环回为例介绍 Spi_demo 项目的分析。

（1）将 CC3200 LaunchPad 的 P2.6（MOSI）和 P2.7（MISO）用短路块连接。

（2）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（3）在 PC 上打开超级终端 Tera Term。

（4）将 spi_demo 项目导入 CCS，Spi_demo 项目包含下列 4 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 和 SPI 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Spi_demo 项目程序流程如图 3.32 所示。

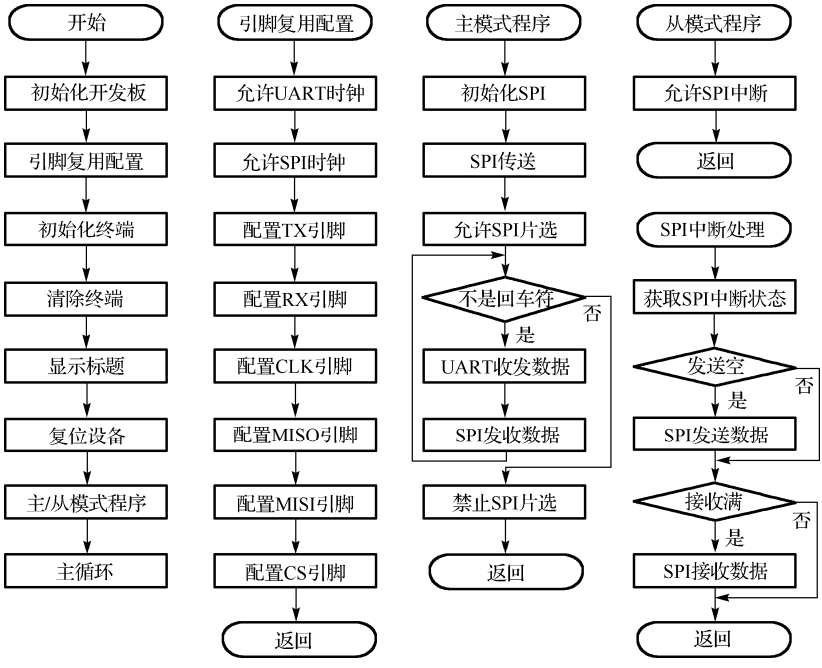


图 3.32　Spi_demo 项目程序流程图

（5）将 main.c 中宏定义 MASTER_MODE 的值从 0 改为 1。

（6）将主模式程序 MasterMain()的“while(ulUserData != '\r')”循环体

```
ulUserData = MAP_UARTCharGet(UARTA0_BASE);          //从 UART 终端读字符
MAP_UARTCharPut(UARTA0_BASE, ulUserData);           //回显字符
MAP_SPIDataPut(GSPI_BASE, ulUserData);              //通过 SPI 发送字符
MAP_SPIDataGet(GSPI_BASE, &ulDummy);                //通过 SPI 接收字符
```

修改为

```
ulUserData = MAP_UARTCharGet(UARTA0_BASE);          //从 UART 终端读字符
MAP_SPIDataPut(GSPI_BASE, ulUserData);              //通过 SPI 发送字符
MAP_SPIDataGet(GSPI_BASE, &ulDummy);                //通过 SPI 接收字符
MAP_UARTCharPut(UARTA0_BASE, ulDummy);              //回显字符
```

（7）进入调试界面运行程序。

（8）在超级终端中按任一键通过 CC3200 GSPI_MOSI 发送字符串“This is CC3200 SPI Master Application”，经 GSPI_MISO 环回接收后再通过 UART 发送到 PC 超级终端显示。

（9）在超级终端中输入“asdfghjkl”等任意字符，输入字符通过 SPI 发送，SPI 环回接收后再通过 UART 回显到超级终端。

Spi_demo 项目的运行结果如图 3.33 所示。

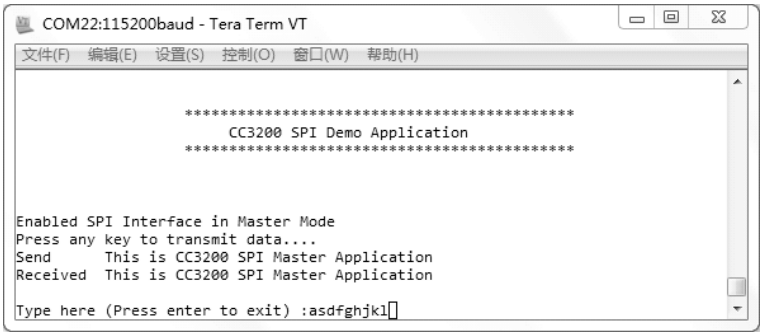


图 3.33　Spi_demo 项目运行结果

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- 终端操作。
- 初始化 SPI。
- SPI 数据操作。
- SPI 中断操作。

1．初始化开发板 BoardInit()

初始化开发板包括设置中断向量表、允许处理器中断、允许 SysTick 中断和初始化 MCU 等步骤，调用了相应的中断 API 函数，具体内容和 Blinky 项目相同。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序由 PinMux 应用程序根据设计需要自动生成，程序代码在 pinmux.c 中，其中调用了相应的 PRCM 和 PIN API 函数。Spi_demo 项目除了使用 SPI 接口外，还要使用 UART 接口在 PC 上显示信息，因此需要对 SPI 和 UART 分别进行配置。

（1）允许外设时钟 PRCMPeripheralClkEnable()。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：外设号，GSPI 的外设号是 0x03，UARTA0 的外设号是 0x0C，都在 prcm.h 中定义。
- ulClkFlags：时钟标志，设置值是 PRCM_RUN_MODE_CLK（运行模式时钟）。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

（2）配置 SPI 引脚 PinTypeSPI()。

```
void PinTypeSPI(unsigned long ulPin, unsigned long ulPinMode);
```

功能：配置 SPI 引脚。

参数说明：

- ulPin：引脚号，GSPI_CLK、GSPI_MISO、GSPI_MOSI 和 GSPI_CS 的引脚号依次是 PIN_05、PIN_06、PIN_07 和 PIN_08。
- ulPinMode：引脚模式，PIN_05～PIN_08 的 PIN_MODE_7 为 GSPI 模式。

PinTypeSPI()的声明在 pin.h 中，代码在 pin.c 中。PinTypeSPI()也通过调用 PinModeSet()和 PinConfigSet()实现引脚类型的配置。

3．终端操作

终端操作包括初始化终端、清除终端、显示标题和显示用法等，其中调用了相应的 UART API 函数。

（1）初始化终端 InitTerm()和清除终端 ClearTerm()的使用方法与 uart_demo 项目相同。

（2）主程序通过调用 Message()显示下列标题。

```
*********************************************************************
                    CC3200 SPI Demo Application
*********************************************************************
```

（3）主模式程序通过调用 Message()和 Report()显示下列用法信息。

```
Enabled SPI Interface in Master Mode          允许主模式 SPI 接口
Press any key to transmit data....            按任一键发送数据
```

按任一键后程序显示下列信息。

```
Send        This is CC3200 SPI Master Application
Received    This is CC3200 SPI Master Application
Type here (Press enter to exit) :                    在这里输入（按回车键退出）
```

从模式程序通过调用 Message()显示下列信息。

```
Enabled SPI Interface in Slave Mode                  允许从模式 SPI 接口
Received :
```

主/从模式程序通过下列宏定义选择。

```
#define MASTER_MODE      0
```

程序默认值是 0，运行从模式程序。作为主设备（Master）使用的程序需将宏定义的值改为 1，运行主模式程序。

4．初始化 SPI

初始化 SPI 主要包括下列内容，其中最主要的是 SPIConfigSetExpClk()。

- 复位外设：PRCMPeripheralReset()。
- 复位 SPI：SPIReset()。
- 配置 SPI 接口：SPIConfigSetExpClk()。
- 允许 SPI：SPIEnable()。

（1）复位外设 PRCMPeripheralReset()

```
void PRCMPeripheralReset(unsigned long ulPeripheral);
```

功能：复位外设。

参数说明：ulPeripheral，外设号，GSPI 的外设号是 0x03，在 prcm.h 中定义。

PRCMPeripheralReset()的声明在 prcm.h 中，代码在 prcm.c 中。

（2）复位 SPI 函数 SPIReset()。

```
void SPIReset(unsigned long ulBase);
```

功能：复位 SPI。

参数说明：ulBase，SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。

SPIReset()的声明在 spi.h 中，代码在 spi.c 中。

（3）配置 SPI 接口 SPIConfigSetExpClk()。

配置 SPI 接口函数 SPIConfigSetExpClk()的函数调用如图 3.34 所示。

```
181    //
182    // Configure SPI interface
183    //
184    MAP_SPIConfigSetExpClk(GSPI_BASE,MAP_PRCMPeripheralClockGet(PRCM_GSPI),
185                           SPI_IF_BIT_RATE,SPI_MODE_MASTER,SPI_SUB_MODE_0,
186                           (SPI_SW_CTRL_CS |
187                           SPI_4PIN_MODE |
188                           SPI_TURBO_OFF |
189                           SPI_CS_ACTIVEHIGH |
190                           SPI_WL_8));
```

图 3.34　Spi_demo 项目 SPIConfigSetExpClk()函数调用

函数首先调用 PRCMPeripheralClkGet()获得 SPI 时钟，然后对 SPI 接口进行配置。SPIConfigSetExpClk()函数的部分内容如图 3.35 所示。

```
741 void
742 SPIConfigSetExpClk(unsigned long ulBase,unsigned long ulSPIClk,
743             unsigned long ulBitRate, unsigned long ulMode,
744             unsigned long ulSubMode, unsigned long ulConfig)
745 {
746
747   unsigned long ulRegData;
748   unsigned long ulDivider;
749
750   //
751   // Read MODULCTRL register
752   //
753   ulRegData = HWREG(ulBase + MCSPI_O_MODULCTRL);
```

图 3.35　Spi_demo 项目 SPIConfigSetExpClk()函数（部分）

```
void SPIConfigSetExpClk(unsigned long ulBase, unsigned long ulSPIClk,
                        unsigned long ulBitRate, unsigned long ulMode,
                        unsigned long ulSubMode, unsigned long ulConfig);
```

功能：配置 SPI 接口。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- ulSPIClk：SPI 时钟（40 MHz），通过调用 PRCMPeripheralClkGet()获得。
- ulBitRate：SPI 比特率（主模式），设置值为 100000。
- ulMode：SPI 模式，包括下列值。
 - ✧ SPI_MODE_MASTER：主模式，0x0000 0000。
 - ✧ SPI_MODE_SLAVE：从模式，0x0000 0004。
- ulSubMode：SPI 子模式，子模式 0～3 分别对应两种时钟极性和两种时钟相位的不同组合，设置值是子模式 0。ulSubMode 包括下列值。
 - ✧ SPI_SUB_MODE_0：子模式 0，0x0000 0000。
 - ✧ SPI_SUB_MODE_1：子模式 1，0x0000 0001。
 - ✧ SPI_SUB_MODE_2：子模式 2，0x0000 0002。
 - ✧ SPI_SUB_MODE_3：子模式 3，0x0000 0003。
- ulConfig：SPI 参数，包括数据位数（8 位、16 位或 32 位）、片选电平（低电平或高电平）、片选方式（软件或硬件）、3/4 线方式和 TURBO 方式等下列值。
 - ✧ SPI_WL_8：8 位数据，0x0000 0380。
 - ✧ SPI_WL_16：16 位数据，0x0000 0780。
 - ✧ SPI_WL_32：32 位数据，0x0000 0F80。
 - ✧ SPI_CS_ACTIVELOW：片选低电平有效，0x0000 0040。
 - ✧ SPI_CS_ACTIVEHIGH：片选高电平有效，0x0000 0000。
 - ✧ SPI_SW_CTRL_CS：软件控制片选，0x0100 0000。
 - ✧ SPI_HW_CTRL_CS：硬件控制片选，0x0000 0000。
 - ✧ SPI_3PIN_MODE：3 线方式，0x0200 0000。
 - ✧ SPI_4PIN_MODE：4 线方式，0x0000 0000。

- ✧ SPI_TURBO_ON：TURBO 方式开启，0x0008 0000。
- ✧ SPI_TURBO_OFF：TURBO 方式关闭，0x0000 0000。

设置值是 SPI_WL_8（8 位数据）、SPI_CS_ACTIVEHIGH（片选高电平有效）、SPI_SW_CTRL_CS（软件控制片选）、SPI_4PIN_MODE （4 线方式）和 SPI_TURBO_OFF （TURBO 方式关闭），参见图 3.34。

SPIConfigSetExpClk()的声明在 spi.h 中，代码在 spi.c 中。

（4）允许 SPI 函数 SPIEnable()。

```
void SPIEnable(unsigned long ulBase);
```

功能：允许 SPI。

参数说明：ulBase，SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。

SPIEnable()的声明在 spi.h 中，代码在 spi.c 中。

5．SPI 数据操作

SPI 数据操作主要包括下列内容，其中最主要的是 SPIDataPut()和 SPIDataGet()。

- SPI 传送：SPITransfer()。
- 允许 SPI 片选：SPICSEnable()。
- SPI 发送数据：SPIDataPut()。
- SPI 接收数据：SPIDataGet()。
- SPI 非阻塞接收数据：SPIDataGetNonBlocking()。
- 禁止 SPI 片选：SPICSDisable()。

（1）SPI 传送 SPITransfer()

```
long SPITransfer(unsigned long ulBase, unsigned char *ucDout,
                 unsigned char *ucDin, unsigned long ulCount,
                 unsigned long ulFlags);
```

功能：通过 SPI 发送和接收数据。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- *ucDout：发送数据缓存区指针或 0。
- *ucDin：接收数据缓存区指针或 0。
- ulCount：数据字节数。
- ulFlags：片选控制标志，包括下列值。
 - ✧ SPI_CS_ENABLE：片选允许，0x0000 0001。
 - ✧ SPI_CS_DISABLE：片选禁止，0x0000 0002。

返回值：0 表示成功，–1 表示其他。

SPITransfer()的声明在 spi.h 中，代码在 spi.c 中。SPITransfer()分别通过调用 SPITransfer8()、SPITransfer16()和 SPITransfer32()实现 8 位、16 位、32 位数据的传送，三者的具体操作和 SPITransfer()类似，不再赘述。

（2）允许 SPI 片选 SPICSEnable()。

```
void SPICSEnable(unsigned long ulBase);
```

功能：允许 SPI 片选。

参数说明：ulBase，SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。

SPICSEnable()的声明在 spi.h 中，代码在 spi.c 中。

（3）SPI 发送数据 SPIDataPut()。

SPIDataPut()函数的内容如图 3.36 所示。

```
937 void
938 SPIDataPut(unsigned long ulBase, unsigned long ulData)
939 {
940   //
941   // Wait for space in FIFO
942   //
943   while(!(HWREG(ulBase + MCSPI_O_CH0STAT)&MCSPI_CH0STAT_TXS))
944   {
945   }
946
947   //
948   // Write the data
949   //
950   HWREG(ulBase + MCSPI_O_TX0) = ulData;
951 }
```

图 3.36　Spi_demo 项目 SPIDataPut()函数

```
void SPIDataPut(unsigned long ulBase, unsigned long ulData);
```

功能：SPI 发送数据。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- ulData：发送数据。

SPIDataPut()的声明在 spi.h 中，代码在 spi.c 中。

（4）SPI 接收数据 SPIDataGet()。

SPIDataGet()函数的内容如图 3.37 所示。

```
872 void
873 SPIDataGet(unsigned long ulBase, unsigned long *pulData)
874 {
875   //
876   // Wait for Rx data
877   //
878   while(!(HWREG(ulBase + MCSPI_O_CH0STAT) & MCSPI_CH0STAT_RXS))
879   {
880   }
881
882   //
883   // Read the value
884   //
885   *pulData = HWREG(ulBase + MCSPI_O_RX0);
886 }
```

图 3.37　Spi_demo 项目 SPIDataGet()函数及其参数说明

```
void SPIDataGet(unsigned long ulBase, unsigned long *pulData);
```

功能：SPI 接收数据。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- *pulData：接收数据指针。

SPIDataGet()的声明在 spi.h 中，代码在 spi.c 中。

（5）SPI 非阻塞接收数据 SPIDataGetNonBlocking()。

```
long SPIDataGetNonBlocking(unsigned long ulBase, unsigned long *pulData);
```

功能：SPI 非阻塞接收数据。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- *pulData：接收数据指针。

返回值：接收数据数，0 或 1。

SPIDataGetNonBlocking()的声明在 spi.h 中，代码在 spi.c 中。

SPI 非阻塞接收数据和 SPI 接收数据的主要区别是：当没有接收数据时，SPI 非阻塞接收数据返回 0，而 SPI 接收数据一直等待（阻塞，参见图 3.37 中的 while 语句），直到接收到数据为止。如果接收数据前已判断接收缓存区满标志，则两者等效。

（6）禁止 SPI 片选 SPICSDisable()。

```
void SPICSDisable(unsigned long ulBase);
```

功能：禁止 SPI 片选。

参数说明：ulBase，SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。

SPICSDisable()的声明在 spi.h 中，代码在 spi.c 中。

6．SPI 中断操作

SPI 从模式程序采用中断方式工作，SPI 中断操作包括下列内容。

- 注册 SPI 中断：SPIIntRegister()。
- 允许 SPI 中断：SPIIntEnable()。
- 获取 SPI 中断状态：SPIIntStatus()。
- 清除 SPI 中断标志：SPIIntClear()。

（1）注册 SPI 中断 SPIIntRegister()。

```
void SPIIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
```

功能：注册 SPI 中断。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- *pfnHandler：SPI 中断处理程序指针，SPI 中断处理程序是 SlaveIntHandler()。

SPIIntRegister()的声明在 spi.h 中，代码在 spi.c 中。SPIIntRegister()通过调用 SPIIntNumberGet()、IntRegister()和 IntEnable()实现中断处理程序的注册。

```
long SPIIntNumberGet(unsigned long ulBase);
```

功能：获取 SPI 中断号。

参数说明：ulBase，SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。

返回值：SPI 中断号（GSPI 中断号是 192，在 hw_ints.h 中定义）或–1（基地址无效）。

SPIIntNumberGet()的代码在 spi.c 中。

```
void IntRegister(unsigned long ulInterrupt, void (*pfnHandler)(void));
```

功能：注册中断处理程序。

参数说明：

- ulInterrupt：中断号，GSPI 中断号是 192，在 hw_ints.h 中定义。
- *pfnHandler：中断处理程序指针。

IntRegister()的声明在 interrupt.h 中，代码在 interrupt.c 中。

```
void IntEnable(unsigned long ulInterrupt);
```

功能：允许中断。

参数说明：ulInterrupt，中断号，GSPI 中断号是 192，在 hw_ints.h 中定义。

IntEnable()的声明在 interrupt.h 中，代码在 interrupt.c 中。

（2）允许 SPI 中断 SPIIntEnable()。

```
void SPIIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
```

功能：允许 SPI 中断。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- ulIntFlags：SPI 中断标志，包括下列值。
 - ✧ SPI_INT_DMATX：DMA 发送中断，0x2000 0000。
 - ✧ SPI_INT_DMARX：DMA 接收中断，0x1000 0000。
 - ✧ SPI_INT_EOW：字计数结束中断，0x0002 0000。
 - ✧ SPI_INT_WKS：唤醒中断，0x0001 0000。
 - ✧ SPI_INT_RX_OVRFLOW：接收溢出中断，0x0000 0008。
 - ✧ SPI_INT_RX_FULL：接收缓存区满中断，0x0000 0004。
 - ✧ SPI_INT_TX_UDRFLOW：发送溢出中断，0x0000 0002。
 - ✧ SPI_INT_TX_ EMPTY：发送缓存区空中断，0x0000 0001。

设置值是 SPI_INT_RX_FULL（接收缓存区满中断）和 SPI_INT_TX_EMPTY（发送缓存区空中断）。SPIIntEnable()的声明在 spi.h 中，代码在 spi.c 中。

（3）获取 SPI 中断状态 SPIIntStatus()。

```
unsigned long SPIIntStatus(unsigned long ulBase, tBoolean bMasked);
```

功能：获取 SPI 中断状态。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- bMasked：中断类型，true-屏蔽中断，false-原始中断。

返回值：SPI 中断状态，值是 5、SPI_INT_RX_FULL（接收缓存区满中断）和 SPI_INT_TX_EMPTY（发送缓存区空中断）。

SPIIntStatus()的声明在 spi.h 中，代码在 spi.c 中。

（4）清除 SPI 中断标志 SPIIntClear()。

```
void SPIIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

功能：清除 SPI 中断标志。

参数说明：

- ulBase：SPI 端口基地址，GSPI 的端口基地址为 0x4402 1000。
- ulIntFlags：SPI 中断标志，值是 5、SPI_INT_RX_FULL（接收缓存区满中断）和 SPI_INT_TX_EMPTY（发送缓存区空中断）。

SPIIntClear()的声明在 spi.h 中，代码在 spi.c 中。

综上所述，Spi_demo 项目使用的与 SPI 相关的 API 函数如表 3.4 所示。

表 3.4　Spi_demo 项目使用的与 SPI 相关的 API 函数

功　能	函数名	调用函数	代码文件
允许外设时钟	**PRCMPeripheralClkEnable()**	PinMuxConfig()	prcm.c
配置 SPI 引脚	**PinTypeSPI()**	PinMuxConfig()	pin.c
设置引脚模式	PinModeSet()	PinTypeSPI()	pin.c
设置引脚类型	PinConfigSet()	PinTypeSPI()	pin.c
复位外设	PRCMPeripheralReset()	main()	prcm.c
复位 SPI	SPIReset()	MasterMain()、SlaveMain()	spi.c
配置 SPI 接口	**SPIConfigSetExpClk()**	MasterMain()、SlaveMain()	spi.c
获得外设时钟	PRCMPeripheralClkGet()	MasterMain()、SlaveMain()	prcm.c
允许 SPI	**SPIEnable()**	MasterMain()、SlaveMain()	spi.c
SPI 传送	SPITransfer()	MasterMain()	spi.c
8 位 SPI 传送	SPITransfer8()	SPITransfer()	spi.c
16 位 SPI 传送	SPITransfer16()	SPITransfer()	spi.c
32 位 SPI 传送	SPITransfer32()	SPITransfer()	spi.c
允许 SPI 片选	**SPICSEnable()**	MasterMain()	spi.c
SPI 发送数据	**SPIDataPut()**	MasterMain()、SlaveIntHandler()	spi.c
SPI 接收数据	**SPIDataGet()**	MasterMain()	spi.c
SPI 非阻塞接收数据	SPIDataGetNonBlocking()	SlaveIntHandler()	spi.c
禁止 SPI 片选	**SPICSDisable()**	MasterMain()	spi.c
注册 SPI 中断程序	SPIIntRegister()	SlaveMain()	spi.c
注册中断处理程序	IntRegister()	SPIIntRegister()	interrupt.c
允许中断	IntEnable()	SPIIntRegister()	interrupt.c
允许 SPI 中断	SPIIntEnable()	SlaveMain()	spi.c
获取 SPI 中断状态	SPIIntStatus()	SlaveIntHandler()	spi.c
清除 SPI 中断标志	SPIIntClear()	SlaveIntHandler()	spi.c

3.3.2 SPI 使用简介

SPI 的使用主要包括引脚复用配置、初始化 SPI、SPI 数据操作和 SPI 中断操作等。

1. 引脚复用配置

引脚复用配置包括允许外设时钟 PRCMPeripheralClkEnable()和配置 SPI 引脚 PinTypeSPI()，API 声明如下。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);

void PinTypeSPI(unsigned long ulPin, unsigned long ulPinMode);
```

2. 初始化 SPI

初始化 SPI 包括配置 SPI 接口 SPIConfigSetExpClk()和允许 SPI SPIEnable()，配置 SPI 接口用于配置 SPI 的比特率、模式和参数，API 声明如下。

```
void SPIConfigSetExpClk(unsigned long ulBase, unsigned long ulSPIClk,
                        unsigned long ulBitRate, unsigned long ulMode,
                        unsigned long ulSubMode, unsigned long ulConfig);

void SPIEnable(unsigned long ulBase);
```

SPIConfigSetExpClk()的 ulMode 参数包含下列值。

- SPI_MODE_MASTER：主模式，0x0000 0000。
- SPI_MODE_SLAVE：从模式，0x0000 0004。

ulSubMode 参数包含下列值。

- SPI_SUB_MODE_0：子模式 0，0x0000 0000。
- SPI_SUB_MODE_1：子模式 1，0x0000 0001。
- SPI_SUB_MODE_2：子模式 2，0x0000 0002。
- SPI_SUB_MODE_3：子模式 3，0x0000 0003。

ulConfig 参数包含下列值。

- SPI_WL_8：8 位数据，0x0000 0380。
- SPI_WL_16：16 位数据，0x0000 0780。
- SPI_WL_32：32 位数据，0x0000 0F80。
- SPI_CS_ACTIVELOW：片选低电平有效，0x0000 0040。
- SPI_CS_ACTIVEHIGH：片选高电平有效，0x0000 0000。
- SPI_SW_CTRL_CS：软件控制片选，0x0100 0000。
- SPI_HW_CTRL_CS：硬件控制片选，0x0000 0000。
- SPI_3PIN_MODE：3 线方式，0x0200 0000。
- SPI_4PIN_MODE：4 线方式，0x0000 0000。
- SPI_TURBO_ON：TURBO 方式开启，0x0008 0000。

- SPI_TURBO_OFF：TURBO 方式关闭，0x0000 0000。

3．SPI 数据操作

SPI 数据操作包括 SPI 传送 SPITransfer()、允许 SPI 片选 SPICSEnable()、SPI 发送数据 SPIDataPut()、SPI 接收数据 SPIDataGet()、SPI 非阻塞接收数据 SPIDataGetNonBlocking()和禁止 SPI 片选 SPICSDisable()等，API 声明如下。

```
long SPITransfer(unsigned long ulBase, unsigned char *ucDout,
                 unsigned char *ucDin, unsigned long ulCount,
                 unsigned long ulFlags);
void SPICSEnable(unsigned long ulBase);
void SPIDataPut(unsigned long ulBase, unsigned long ulData);
void SPIDataGet(unsigned long ulBase, unsigned long *pulData);
long SPIDataGetNonBlocking(unsigned long ulBase, unsigned long *pulData);
void SPICSDisable(unsigned long ulBase);
```

4．SPI 中断操作

SPI 中断操作包括中断初始化和中断处理。中断初始化包括注册 SPI 中断 SPIInt Register()和允许 SPI 中断 SPIIntEnable()，中断处理包括获取 SPI 中断状态 SPIIntStatus()和清除 SPI 中断标志 SPIIntClear()，API 声明如下。

```
void SPIIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void SPIIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long SPIIntStatus(unsigned long ulBase, tBoolean bMasked);
void SPIIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

SPIIntEnable()和 SPIIntClear()的参数 ulIntFlags 包括下列值。

- SPI_INT_DMATX：DMA 发送中断，0x2000 0000。
- SPI_INT_DMARX：DMA 接收中断，0x1000 0000。
- SPI_INT_EOW：字计数结束中断，0x0002 0000。
- SPI_INT_WKS：唤醒中断，0x0001 0000。
- SPI_INT_RX_OVRFLOW：接收溢出中断，0x0000 0008。
- SPI_INT_RX_FULL：接收缓存区满中断，0x0000 0004。
- SPI_INT_TX_UDRFLOW：发送溢出中断，0x0000 0002。
- SPI_INT_TX_ EMPTY：发送缓存区空中断，0x0000 0001。

3.4　I2C 接口应用

内部集成电路总线接口 I2C 是通信控制领域广泛采用的一种总线标准，用于连接微控制器和外围设备，连接在总线上的每个设备都有唯一的 7 位或 10 位地址。

I2C 使用一根双向串行数据线 SDA 和一根双向串行时钟线 SCL 实现主/从设备间的多主串行通信，SDA 和 SCL 的时序关系如图 3.38 所示。

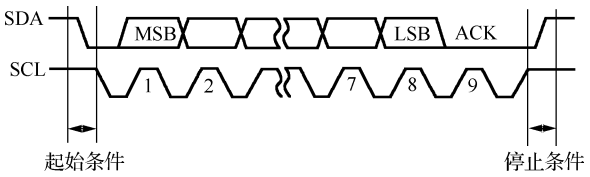


图 3.38　SDA 和 SCL 时序关系图

起始条件是在 SCL 高电平时 SDA 从高电平变为低电平，停止条件是在 SCL 高电平时 SDA 从低电平变为高电平。SDA 上的数据必须在 SCL 高电平时保持稳定，低电平时可以改变。发送器发送数据后释放 SDA（高电平），接收器接收数据后必须在 SCL 低电平时将 SDA 变为低电平，并在 SCL 高电平时保持稳定，作为对发送器的应答（ACK）。

通过 I2C 接口对 I2C 器件进行数据读写的一般格式如图 3.39 所示。

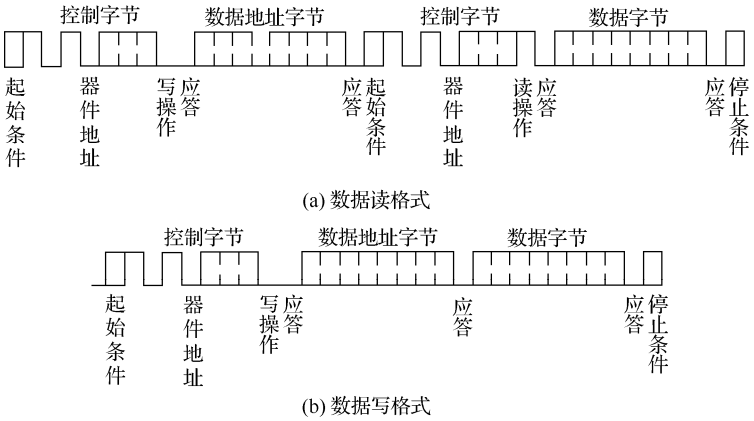


图 3.39　I2C 数据读写一般格式

读数据时，控制器的操作包含两步：写数据地址和读数据。在写数据地址和读数据前，控制器首先发送控制字节（包含 7 位器件地址和 1 位读写操作）。写数据地址前控制字节的最后 1 位为 0（写操作），读数据前控制字节的最后 1 位为 1（读操作）。写数据时，写数据地址和写数据一起进行，控制字节的最后 1 位为 0（写操作）。

CC3200 包含一个 I2C 接口，具有下列特性。

- 可工作在主模式和从模式。
- 主模式具有仲裁和时钟同步。
- 支持多主模式。
- 7 位寻址。
- 标准模式（100 kbps）和快速模式（400 kbps）。

3.4.1　I2C 示例程序分析

I2C 示例程序在“example\i2c_demo”目录中，I2c_demo 项目 CC3200 LaunchPad 电路如图 3.40 所示。

CC3200 通过 I2C 接口与温度传感器 TMP006（U6）和加速度传感器 BMA222（U10）进行通信，两者的器件地址分别是 0x41 和 0x18。

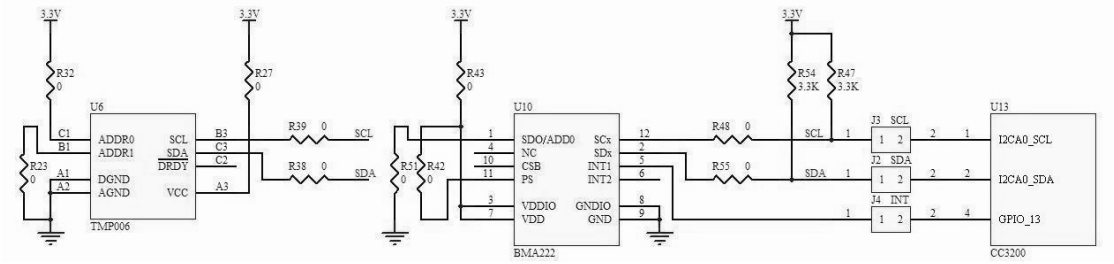


图 3.40　I2c_demo 项目 CC3200 LaunchPad 电路图

（1）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（2）在 PC 上打开超级终端 Tera Term。

（3）将 i2c_demo 项目导入 CCS，i2c_demo 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 和 I2C 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- i2c_if.c：I2C 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

I2c_demo 项目程序流程图如图 3.41 所示。

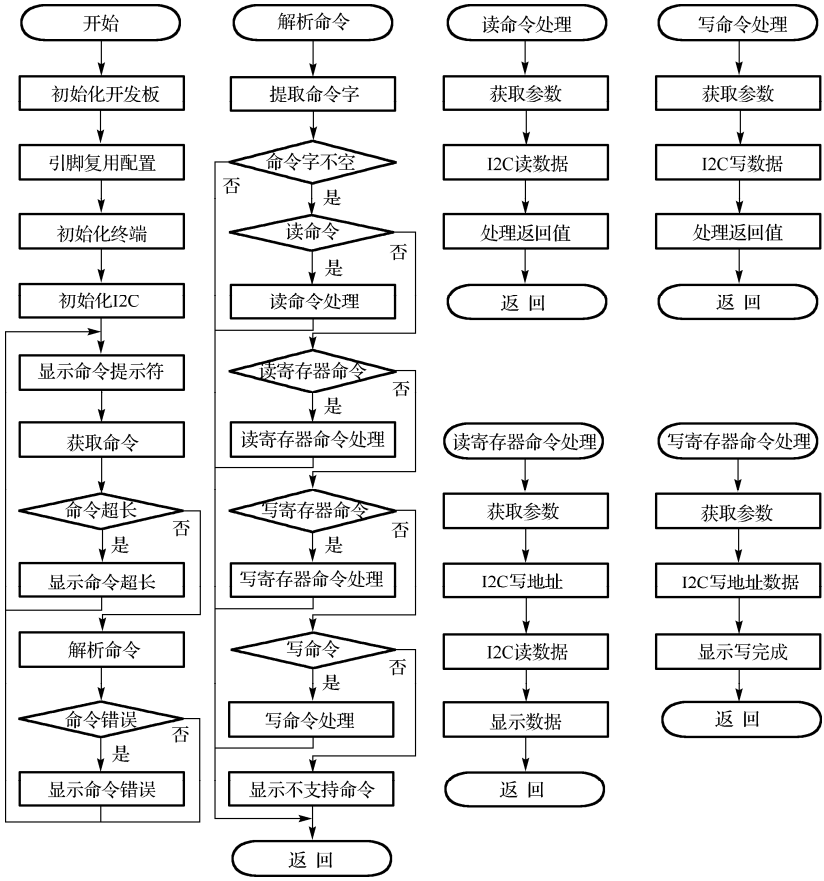


图 3.41　I2c_demo 项目程序流程图

（4）进入调试界面运行程序。

（5）在命令提示符后输入下列命令。

```
readreg 0x41 0xfe 2
```

回车后显示下列信息。

```
I2C Read From address complete:                    从 I2C 器件指定寄存器读数据完成
Read contents                                      读内容
0x54,  0x49,
```

（6）在命令提示符后输入下列命令。

```
read 0x18 1
```

回车后显示下列信息。

```
I2C Read complete                                  从指定 I2C 器件读数据完成
Read contents                                      读内容
0xf8,
```

I2c_demo 项目的运行结果如图 3.42 所示。

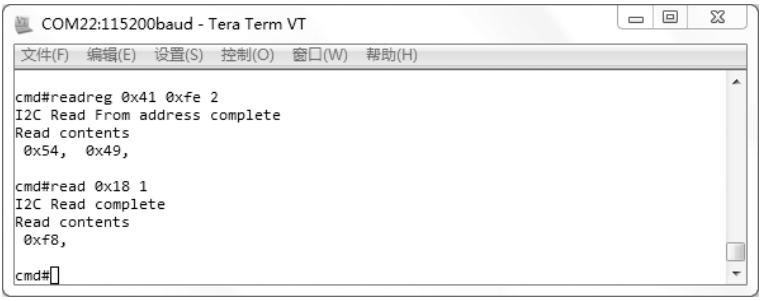


图 3.42　I2c_demo 项目运行结果

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- 终端操作。
- 初始化 I2C。
- I2C 操作。

1．初始化开发板 BoardInit()

初始化开发板包括设置中断向量表、允许处理器中断、允许 SysTick 中断和初始化 MCU 等步骤，调用了相应的中断 API 函数，具体内容和 Blinky 项目相同。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序是由 PinMux 应用程序根据设计需要自动生成的，程序代码在 pinmux.c 中，其中调用了相应的 PRCM 和 PIN API 函数。

I2c_demo 项目除了使用 I2C 接口外，还要使用 UART 接口在 PC 上显示信息，因此需要对 I2C 和 UART 分别进行配置。

（1）允许外设时钟 PRCMPeripheralClkEnable()。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：外设号，I2CA0 和 UARTA0 的外设号分别是 0x14 和 0x0C。
- ulClkFlags：时钟标志，设置值是 PRCM_RUN_MODE_CLK（运行模式时钟）。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

（2）配置 I2C 引脚 PinTypeI2C()。

```
void PinTypeI2C(unsigned long ulPin, unsigned long ulPinMode);
```

功能：配置 I2C 引脚。

参数说明：

- ulPin：引脚号，I2CA0_SCL、和 I2CA0_SDA 的引脚号依次是 PIN_01 和 PIN_02。
- ulPinMode：引脚模式，PIN_01 和 PIN_02 的 PIN_MODE_1 为 I2C 模式。

PinTypeI2C()的声明在 pin.h 中，代码在 pin.c 中。PinTypeI2C()也通过调用 PinModeSet()和 PinConfigSet()实现引脚类型的配置。

3．终端操作

终端操作包括初始化终端、显示标题、显示用法、显示命令提示符和获取命令等。

（1）初始化终端 InitTerm()的使用方法与 uart_demo 项目相同。

（2）显示标题 DisplayBanner()通过调用 Report()显示下列标题。

```
******************************************************************************
                        CC3200 I2C Demo Application
******************************************************************************
```

（3）显示用法 DisplayUsage()通过调用 UART_PRINT()即 Report()显示下列命令用法。

```
Command Usage                                          命令用法
------------------------------------------------------------------------------
write <dev_addr> <wrlen> <<byte0> [<byte1> ... ]> <stop>
       - Write data to the specified i2c device
                                                       向指定 I2C 器件写数据
read  <dev_addr> <rdlen>
       - Read data from the specified i2c device
                                                       从指定 I2C 器件读数据
writereg <dev_addr> <reg_offset> <wrlen> <<byte0> [<byte1> ... ]>
       - Write data to the specified register of the i2c device
                                                       向 I2C 器件指定寄存器写数据
```

```
readreg <dev_addr> <reg_offset> <rdlen>
       - Read data from the specified register of the i2c device
                                                从 I2C 器件指定寄存器读数据
Parameters                                      参数
--------------------------------------------------------------------------
dev_addr - slave address of the i2c device, a hex value preceeded by '0x'
                                    I2C 器件地址，以 0x 开头的十六进制值
reg_offset-register address in the i2c device, a hex value preceeded by '0x'
                                    I2C 器件寄存器地址，以 0x 开头的十六进制值
wrlen - number of bytes to be written, a decimal value
                                                写字节数，十进制值
rdlen - number of bytes to be read, a decimal value
                                                读字节数，十进制值
bytex - value of the data to be written, a hex value preceeded by '0x'
                                                写数据，以 0x 开头的十六进制值
stop - number of stop bits, 0 or 1              停止位数，0 或 1
--------------------------------------------------------------------------
```

（4）显示命令提示符 DisplayPrompt()通过调用 UART_PRINT()即 Report()显示提示符“cmd#”。

（5）获取命令 GetCmd()通过终端获取并回显用户输入的命令。

主程序获取命令后调用解析命令函数 ParseNProcessCmd()解析命令，解析命令首先从命令行提取命令字，然后根据命令字的不同分别调用读命令处理 ProcessReadCommand()、写命令处理 ProcessWriteCommand()、读寄存器命令处理 ProcessReadRegCommand()和写寄存器命令处理 ProcessWriteRegCommand()等处理命令。

4 个命令处理函数首先从命令行提取参数，然后分别调用 I2C 读函数 I2C_IF_Read()和 I2C 写函数 I2C_IF_Write()实现相应的操作。

读写命令和读写寄存器命令的区别是：读写命令从指定器件的当前寄存器读写数据，而读写寄存器命令从器件的指定寄存器读写数据。

读写命令不需要寄存器地址，操作简单；读写寄存器命令需要寄存器地址，操作灵活。

4．初始化 I2C

初始化 I2C 用 I2C 打开 I2C_IF_Open()实现，I2C_IF_Open()的部分内容如图 3.43 所示。

```
364 int
365 I2C_IF_Open(unsigned long ulMode)
366 {
367     //
368     // Enable I2C Peripheral
369     //
370     //MAP_HwSemaphoreLock(HWSEM_I2C, HWSEM_WAIT_FOR_EVER);
371     MAP_PRCMPeripheralClkEnable(PRCM_I2CA0, PRCM_RUN_MODE_CLK);
372     MAP_PRCMPeripheralReset(PRCM_I2CA0);
```

图 3.43　I2c_demo 项目 I2C_IF_Open()函数（部分）

```
int I2C_IF_Open(unsigned long ulMode);
```

功能：打开 I2C 接口。

参数说明：ulMode，I2C 配置模式，包括下列值。

- I2C_MASTER_MODE_STD：标准模式，0。
- I2C_MASTER_MODE_FST：快速模式，1。

返回值：0 表示成功，负值表示失败。

I2C_IF_Open()包括允许外设时钟 PRCMPeripheralClkEnable()、复位外设 PRCMPeripheralReset()和配置 I2C 接口 I2CMasterInitExpClk()，其中主要是配置 I2C 接口 I2CMasterInitExpClk()。

```
void I2CMasterInitExpClk(uint32_t ui32Base, uint32_t ui32I2CClk,
                         bool bFast);
```

功能：配置 I2C 接口。

参数说明：

- ui32Base：I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。
- ui32I2CClk：I2C 时钟，快速传送时钟是 400 kHz，非快速传送时钟是 100 kHz。
- bFast：快速传送标志，true 表示快速传送，false 表示非快速传送。

I2CMasterInitExpClk()的声明在 i2c.h 中，代码在 i2c.c 中。在通过 I2CMasterInitExpClk()配置 I2C 接口前，需要允许 I2C。

```
void I2CMasterEnable(uint32_t ui32Base);
```

功能：允许 I2C。

参数说明：ui32Base，I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。

I2CmasterEnable()的声明在 i2c.h 中，代码在 i2c.c 中。

5．I2C 操作

I2C 操作包括 I2C 写 I2C_IF_Write()和 I2C 读 I2C_IF_Read()，程序代码在 i2c_if.c 中，其中调用了相应的 I2C API 函数。

（1）I2C 写 I2C_IF_Write()。

I2C_IF_Write()的部分内容如图 3.44 所示。

```
159 int
160 I2C_IF_Write(unsigned char ucDevAddr,
161          unsigned char *pucData,
162          unsigned char ucLen,
163          unsigned char ucStop)
164 {
165     RETERR_IF_TRUE(pucData == NULL);
166     RETERR_IF_TRUE(ucLen == 0);
167     //
168     // Set I2C codec slave address
169     //
170     MAP_I2CMasterSlaveAddrSet(I2C_BASE, ucDevAddr, false);
171     //
172     // Write the first byte to the controller.
173     //
174     MAP_I2CMasterDataPut(I2C_BASE, *pucData);
175     //
176     // Initiate the transfer.
177     //
178     RET_IF_ERR(I2CTransact(I2C_MASTER_CMD_BURST_SEND_START));
```

图 3.44　I2c_demo 项目 I2C_IF_Write()函数（部分）

```
int I2C_IF_Write(unsigned char ucDevAddr, unsigned char *pucData,
                 unsigned char ucLen, unsigned char ucStop);
```

功能：I2C 写。

参数说明：

- ucDevAddr：器件地址，温度传感器 TMP006 和加速度传感器 BMA222 的器件地址分别是 0x41 和 0x18。
- *pucData：写数据指针。
- ucLen：写数据长度。
- ucStop：停止位数，0 表示无停止位，1 表示有停止位。

返回值：0 表示成功，负值表示失败。

I2C_IF_Write()通过调用 I2CMasterSlaveAddrSet()、I2CMasterDataPut()和 I2CTransact()实现 I2C 写。

```
void I2CMasterSlaveAddrSet(uint32_t ui32Base, uint8_t ui8SlaveAddr,
                           bool bReceive);
```

功能：设置从地址。

参数说明：

- ui32Base：I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。
- ui8SlaveAddr：7 位从地址，温度传感器 TMP006 和加速度传感器 BMA222 的地址分别是 0x41 和 0x18。
- bReceive：接收标志，true 表示读，false 表示写。

I2CMasterSlaveAddrSet()的声明在 i2c.h 中，代码在 i2c.c 中。

```
void I2CMasterDataPut(uint32_t ui32Base, uint8_t ui8Data);
```

功能：I2C 发送数据。

参数说明：

- ui32Base：I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。
- ui8Data：发送数据。

I2CMasterDataPut()的声明在 i2c.h 中，代码在 i2c.c 中。I2CTransact()在后文中详细介绍。

（2）I2C 读 I2C_IF_Read()。

I2C_IF_Read()的部分内容如图 3.45 所示。

```
232 int
233 I2C_IF_Read(unsigned char ucDevAddr,
234         unsigned char *pucData,
235         unsigned char ucLen)
236 {
237     unsigned long ulCmdID;
238
239     RETERR_IF_TRUE(pucData == NULL);
240     RETERR_IF_TRUE(ucLen == 0);
241     //
242     // Set I2C codec slave address
243     //
244     MAP_I2CMasterSlaveAddrSet(I2C_BASE, ucDevAddr, true);
```

图 3.45　I2c_demo 项目 I2C_IF_Read()函数（部分）

```
int I2C_IF_Read(unsigned char ucDevAddr, unsigned char *pucData,
                unsigned char ucLen);
```

功能：I2C 读。

参数说明：

- ucDevAddr：器件地址，温度传感器 TMP006 和加速度传感器 BMA222 的器件地址分别是 0x41 和 0x18。
- *pucData：读数据指针。
- ucLen：读数据长度。

返回值：0 表示成功，负值表示失败。

I2C_IF_Read()通过调用 I2CMasterSlaveAddrSet()、I2CTransact()和 I2CMasterDataGet()实现 I2C 读。

```
uint32_t I2CMasterDataGet(uint32_t ui32Base);
```

功能：I2C 接收数据。

参数说明：ui32Base，I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。

返回值：接收数据。

I2CMasterDataGet()的声明在 i2c.h 中，代码在 i2c.c 中。

（3）I2C 处理 I2CTransact()。

```
static int I2CTransact(unsigned long ulCmd);
```

功能：I2C 处理。

参数说明：ulCmd，处理命令，主要包括下列值。

- I2C_MASTER_CMD_SINGLE_SEND：单字节发送，0x0000 0007。
- I2C_MASTER_CMD_SINGLE_RECEIVE：单字节接收，0x0000 0007。
- I2C_MASTER_CMD_BURST_SEND_START：突发发送开始，0x0000 0003。
- I2C_MASTER_CMD_BURST_SEND_CONT：突发发送继续，0x0000 0001。
- I2C_MASTER_CMD_BURST_SEND_STOP：突发发送停止，0x0000 0004。
- I2C_MASTER_CMD_BURST_RECEIVE_START：突发接收开始，0x0000 000b。
- I2C_MASTER_CMD_BURST_RECEIVE_CONT：突发接收继续，0x0000 0009。
- I2C_MASTER_CMD_BURST_RECEIVE_FINISH：突发接收完成，0x0000 0005。

返回值：0 表示成功，负值表示失败。

I2CTransact()通过调用 I2CMasterIntClearEx()、I2CMasterIntStatusEx、I2CMasterTimeoutSet()、I2CMasterControl()和 I2CMasterErr()等实现 I2C 处理。

```
void I2CMasterIntClearEx(uint32_t ui32Base, uint32_t ui32IntFlags);
```

功能：清除 I2C 中断标志。

参数说明：

- ui32Base：I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。

- ui32IntFlags：I2C 中断标志，通过调用 I2CMasterIntStatusEx()获得。

I2CMasterIntClearEx()的声明在 i2c.h 中，代码在 i2c.c 中。

```
uint32_t I2CMasterIntStatusEx(uint32_t ui32Base, bool bMasked);
```

功能：获取 I2C 中断状态。

参数说明：

- ui32Base：I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。
- bMasked：中断类型，true 表示屏蔽中断，false 表示原始中断。

返回值：当前中断状态。

I2CMasterIntStatusEx()的声明在 i2c.h 中，代码在 i2c.c 中。

```
void I2CMasterTimeoutSet(uint32_t ui32Base, uint32_t ui32Value);
```

功能：设置 I2C 超时。

参数说明：

- ui32Base：I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。
- ui32Value：超时值。

I2CMasterTimeoutSet()的声明在 i2c.h 中，代码在 i2c.c 中。

```
void I2CMasterControl(uint32_t ui32Base, uint32_t ui32Cmd);
```

功能：控制 I2C。

参数说明：

- ui32Base：I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。
- ui32Cmd：处理命令。

I2CMasterControl()的声明在 i2c.h 中，代码在 i2c.c 中。

```
uint32_t I2CMasterErr(uint32_t ui32Base);
```

功能：获取 I2C 错误状态。

参数说明：ui32Base，I2C 端口基地址，I2CA0 的端口基地址为 0x4002 0000。

返回值：错误状态。

I2CMasterControl()的声明在 i2c.h 中，代码在 i2c.c 中。

综上所述，I2c_demo 项目使用的与 I2C 有关的 API 函数如表 3.5 所示。

表 3.5　I2c_demo 项目使用的与 I2C 有关的 API 函数

功　能	函数名	调用函数	代码文件
允许外设时钟	**PRCMPeripheralClkEnable()**	PinMuxConfig()	prcm.c
配置 I2C 引脚	**PinTypeI2C()**	PinMuxConfig()	pin.c
设置引脚模式	PinModeSet()	PinTypeI2C()	pin.c
设置引脚类型	PinConfigSet()	PinTypeI2C()	pin.c
I2C 打开	**I2C_IF_Open()**	main()	i2c_if.c

续表

功　能	函数名	调用函数	代码文件
复位外设	PRCMPeripheralReset()	I2C_IF_Open()	prcm.c
配置 I2C 接口	I2CMasterInitExpClk()	I2C_IF_Open()	i2c.c
允许 I2C	I2CMasterEnable()	I2CMasterInitExpClk()	i2c.c
I2C 写	**I2C_IF_Write()**	ProcessReadRegCommand()、ProcessWriteRegCommand()、ProcessWriteCommand()	i2c_if.c
I2C 读	**I2C_IF_Read()**	ProcessReadCommand()、ProcessReadRegCommand()	i2c_if.c
设置从地址	I2CMasterSlaveAddrSet()	I2C_IF_Read()、I2C_IF_Write()	i2c.c
I2C 接收数据	I2CMasterDataGet()	I2C_IF_Read()、	i2c.c
I2C 发送数据	I2CMasterDataPut()	I2C_IF_Write()	i2c.c
I2C 处理	I2CTransact()	I2C_IF_Read()、I2C_IF_Write()	i2c_if.c
清除 I2C 中断标志	I2CMasterIntClearEx()	I2CTransact()	i2c.c
获取 I2C 中断状态	I2CMasterIntStatusEx()	I2CTransact()	i2c.c
设置 I2C 超时	I2CMasterTimeoutSet()	I2CTransact()	i2c.c
控制 I2C	I2CMasterControl()	I2CTransact()	i2c.c
获取 I2C 错误状态	I2CMasterErr()	I2CTransact()	i2c.c

3.4.2　I2C 使用简介

I2C 的使用主要包括引脚复用配置、初始化 I2C 和 I2C 操作等。

1．引脚复用配置

引脚复用配置包括允许外设时钟 PRCMPeripheralClkEnable()和配置 I2C 引脚 PinType I2C()，API 声明如下。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);

void PinTypeI2C(unsigned long ulPin, unsigned long ulPinMode);
```

2．初始化 I2C

初始化 I2C 用 I2C 打开 I2C_IF_Open()实现，主要是配置 I2C 接口 I2CMasterInitExpClk()，用于配置 I2C 时钟，其中包含允许 I2C I2CMasterEnable()，API 声明如下。

```
int I2C_IF_Open(unsigned long ulMode);

void I2CMasterInitExpClk(uint32_t ui32Base, uint32_t ui32I2CClk,
                         bool bFast);

void I2CMasterEnable(uint32_t ui32Base);
```

3．I2C 操作

I2C 操作包括 I2C 写 I2C_IF_Write()和 I2C 读 I2C_IF_Read()，分别通过调用设置从地址 I2CMasterSlaveAddrSet()、I2C 发送数据 I2CMasterDataPut()、I2C 接收数据 I2CMasterDataGet()和 I2C 处理 I2CTransact()实现，API 声明如下。

```
int I2C_IF_Write(unsigned char ucDevAddr, unsigned char *pucData,
                 unsigned char ucLen, unsigned char ucStop);

int I2C_IF_Read(unsigned char ucDevAddr, unsigned char *pucData,
                unsigned char ucLen);

void I2CMasterSlaveAddrSet(uint32_t ui32Base, uint8_t ui8SlaveAddr,
                           bool bReceive);

void I2CMasterDataPut(uint32_t ui32Base, uint8_t ui8Data);

uint32_t I2CMasterDataGet(uint32_t ui32Base);

static int I2CTransact(unsigned long ulCmd);
```

注意：I2C 的使用比 UART 和 SPI 复杂，通常不像 UART 和 SPI 那样直接使用配置 I2C 接口 I2CMasterInitExpClk()、I2C 发送数据 I2CMasterDataPut()和 I2C 接收数据 I2CMaster DataGet()等进行初始化和读写数据，而是使用 I2C 打开 I2C_IF_Open()、I2C 读 I2C_IF_Read()和 I2C 写 I2C_IF_Write()等进行初始化和读写数据。

3.4.3 温度测量和加速度测量程序设计

下面以温度测量和加速度测量为例，对 I2C 示例程序进行改进。

1．温度测量程序设计

温度测量使用温度传感器 TMP006 实现，TMP006 的内部寄存器保存有温度测量结果（–40℃～125℃）、配置和状态信息，CC3200 通过 I2C 接口可以读写这些信息。TMP006 的内部寄存器如表 3.6 所示。

表 3.6 TMP006 内部寄存器

地址	名 称	类型	复位值	说 明
0x00	V_{OBJECT}	只读	0x0000	传感器电压寄存器（15:0～16 位补码表示）
0x01	$T_{AMBIENT}$	只读	0x0000	温度寄存器（15:2～14 位补码表示）
0x02	Configuratiom	读/写	0x7480	配置寄存器
0xFE	Manufacturer ID	只读	0x5449	制造商标识寄存器
0xFF	Device ID	只读	0x0067	器件标识寄存器

TMP006 的温度值用 14 位（15:2）补码表示，第 15 位是符号位（0 表示正，1 表示负），第 14:7 位是 8 位整数部分，第 6:2 位是 5 位小数部分，最小分辨率是 2^{-5}=0.03125。

温度测量程序在 I2C 示例程序的基础上做如下修改。

（1）在 main()中将语句

```
char acCmdStore[512];
```

修改为

```
unsigned char ucData[2];
```

（2）在 main()中 while(FOREVER)语句前添加下列语句。

```
ucData[0] = 1;
I2C_IF_Write(0x41, &ucData[0], 1, 0);                    //地址指向温度寄存器
```

（3）将 while(FOREVER)循环体替换为

```
iRetVal = I2C_IF_Read(0x41, &ucData[0], 2);              //读取温度值
if(!iRetVal)                                             //成功，处理并显示温度值
{
    unsigned short usTemp = (ucData[0]<<8) + ucData[1];
    if(!(usTemp & 0x8000))                               //处理正温度值
    {
        Report("当前温度是: %d.", usTemp>>7);            //显示整数部分
        Report("%02d℃\n\r", ((usTemp&0x7f)*25)>>5);//显示小数部分（两位）
    }
}
MAP_UtilsDelay(8000000);                                 //延时约 0.6s
```

运行程序，在 PC 超级终端上将显示温度值，用手触摸实验板上的温度传感器 U6，显示的温度值发生变化，如图 3.46 所示。

当前温度是：27.68℃
当前温度是：28.86℃
当前温度是：29.59℃

图 3.46　温度测量结果

2．加速度测量程序设计

加速度测量使用加速度传感器 BMA222 实现，BMA222 的内部寄存器有 64 个，CC3200 通过 I2C 或 SPI 接口可以读写这些寄存器。常用的 BMA222 内部寄存器如表 3.7 所示。

表 3.7　常用 BMA222 内部寄存器

地址	名　　称	类型	复位值	说　　明
0x00	Chip ID	只读	0xF8	芯片标识寄存器
0x02	new_data_x	只读	0x00	x 轴新数据标志寄存器
0x03	acc_x	只读	0x00	x 轴加速度数据寄存器
0x04	new_data_y	只读	0x00	y 轴新数据标志寄存器
0x05	acc_y	只读	0x00	y 轴加速度数据寄存器
0x06	new_data_z	只读	0x00	z 轴新数据标志寄存器
0x07	acc_z	只读	0x00	z 轴加速度数据寄存器

加速度测量程序在 I2C 示例程序的基础上做如下修改。

（1）在 main()中将语句

```
char acCmdStore[512];
```

修改为

```
unsigned char ucData[6];
```

（2）在 main()中 while(FOREVER)语句前添加下列语句。

```
ucData[0] = 2;
I2C_IF_Write(0x18, &ucData[0], 1, 0);          //地址指向加速度数据寄存器
```

（3）将 while(FOREVER)循环体替换为下列语句。

```
iRetVal = I2C_IF_Read(0x18, &ucData[0], 6);    //读取加速度值
if(!iRetVal)                                   //成功，显示加速度值
{
   Report("x: %03d  ", ucData[1]);             //显示 x 轴加速度
   Report("y: %03d  ", ucData[3]);             //显示 y 轴加速度
   Report("z: %03d\n\r", ucData[5]);           //显示 z 轴加速度
}
MAP_UtilsDelay(8000000);                       //延时约 0.6s
```

运行程序，在 PC 超级终端上将显示加速度值，移动实验板，显示的加速度值发生变化，如图 3.47 所示。

```
x: 033  y: 197  z: 248
x: 253  y: 194  z: 004
x: 253  y: 193  z: 003
```

图 3.47　加速度测量结果

第 4 章

片内外设应用

CC3200 支持下列片内外设。

- 通用定时器（GDT）。
- 模/数转换器（ADC）。
- 嵌套向量中断控制器（NVIC）。
- 微型直接存储器存取（μDMA）。

本章主要在分析片内外设示例程序的基础上，介绍片内外设的使用，并对示例程序进行修改，完善示例程序的功能。

4.1 GPT 应用

CC3200 包含 4 个 32 位用户可编程通用定时器（GPTA0～3 或 TIMERA0～3），GPT 可以对定时器输入引脚的外部事件进行计数或定时。每个 GPT 模块包含 2 个 16 位定时/计数器（TimerA 和 TimerB），可以作为定时器或事件计数器独立工作，也可以作为一个 32 位定时器工作。GPT 模块具有下列特性。

- 操作模式。
 - ✧ 16 位或 32 位可编程单次定时器。
 - ✧ 16 位或 32 位可编程周期定时器。
 - ✧ 16 位通用定时器，带 8 位预分频器。
 - ✧ 16 位输入边沿计数或时间捕捉模式，带 8 位预分频器。
 - ✧ 16 位脉冲宽度调制（PWM）模式，带 8 位预分频器和软件可编程输出反相。
- 向上或向下计数。
- 16 个 16 位或 32 位捕捉比较 PWM（CCP）引脚。
- 可以确定产生定时器中断到进入中断服务程序（ISR）的时间。
- 可以触发使用 DMA 的高效传输。
- 系统时钟运行（80 MHz）。

通用定时器的示例程序有 3 个：定时器示例程序、PWM 示例程序和计数捕捉示例程序，下面分别对这 3 个示例程序进行分析。

4.1.1 定时器示例程序分析

定时器示例程序在“example\timer”目录中，Timer 项目 CC3200 LaunchPad 电路如图 4.1 所示。

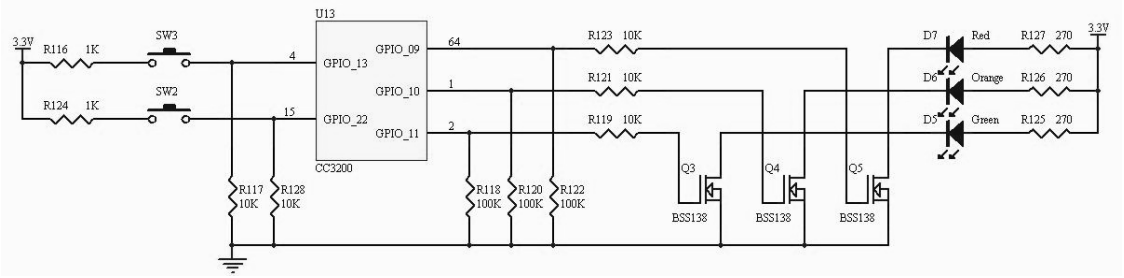


图 4.1 Timer 项目 CC3200 LaunchPad 电路图

CC3200 GPTA0A 和 GPTA1A 分别通过 GPIO_09（PIN_64）和 GPIO_11（PIN_02）控制红 LED 和绿 LED 的闪烁时间，红 LED 的闪烁时间是 2 s，绿 LED 的闪烁时间是 1 s。

Timer 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO 引脚配置程序文件。
- timer_if.c：Timer 驱动程序文件。
- gpio_if.c：GPIO 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Timer 项目程序流程图如图 4.2 所示。

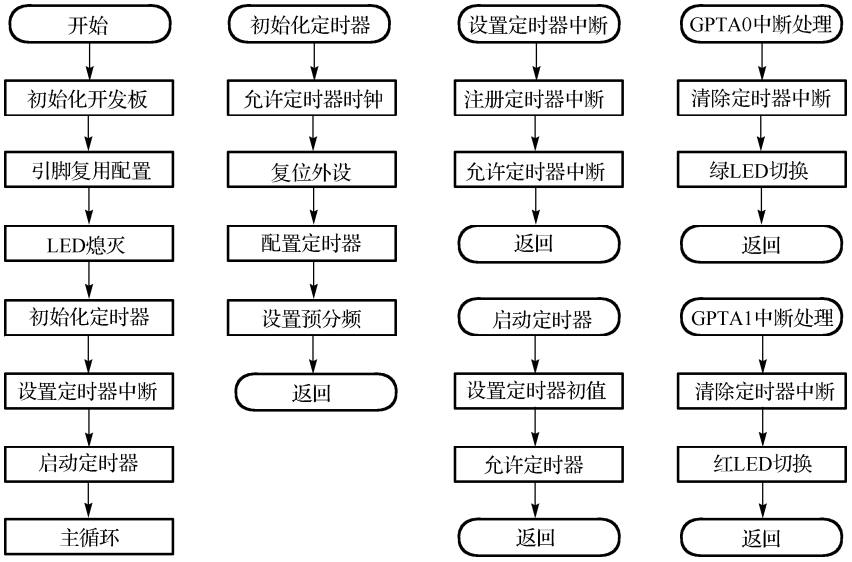


图 4.2 Timer 项目程序流程图

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- 定时器操作。
- 定时器中断处理。

1．初始化开发板 BoardInit()

初始化开发板包括设置中断向量表、允许处理器中断、允许 SysTick 中断和初始化 MCU 等步骤，调用了相应的中断 API 函数，具体内容和 Blinky 项目相同。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序由 PinMux 应用程序根据设计需要自动生成，程序代码在 pinmux.c 中，其中调用了相应的 PRCM 和 PIN API 函数。

Timer 项目的引脚复用配置和 Blinky 项目类似，主要包括允许 GPIOA1 时钟、配置红 LED 引脚 PIN_64 和绿 LED 引脚 PIN_02。

3．定时器操作

定时器操作包括下列内容。

- 初始化定时器：Timer_IF_Init()（TimerConfigure()）。
- 设置定时器中断：Timer_IF_IntSetup()。
- 启动定时器：Timer_IF_Start()（TimerLoadSet()）。

程序代码在 timer_if.c 中，其中调用了相应的 Timer API 函数，主要是 TimerConfigure() 和 TimerLoadSet()等。

（1）初始化定时器 Timer_IF_Init()。

```
void Timer_IF_Init(unsigned long ePeripheral, unsigned long ulBase,
                   unsigned long ulConfig, unsigned long ulTimer,
                   unsigned long ulValue);
```

功能：初始化定时器。

参数说明：

- ePeripheral：外设号，TimerA0 和 TimerA1 的外设号分别是 0x0E 和 0x0F，在 prcm.h 中定义。
- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulConfig：定时器配置，TimerA0 和 TimerA1 的设置值都是 TIMER_CFG_PERIODIC（32 位周期定时器）。
- ulTimer：定时器选择，TimerA0 和 TimerA1 的设置值都是 Timer_A（定时器 A）。
- ulValue：预分频值，0～255（1～256 分频），TimerA0 和 TimerA1 的设置值都是 0（1 分频即不分频）。

Timer_IF_Init()依次调用 PRCMPeripheralClkEnable()、PRCMPeripheralReset()、TimerConfigure()和 TimerPrescaleSet()等实现定时器初始化。

① 允许定时器时钟 PRCMPeripheralClkEnable()。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：外设号，TimerA0 和 TimerA1 的外设号分别是 0x0E 和 0x0F。
- ulClkFlags：时钟标志，设置值是 PRCM_RUN_MODE_CLK（运行模式时钟）。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

② 复位定时器 PRCMPeripheralReset()。

```
void PRCMPeripheralReset(unsigned long ulPeripheral);
```

功能：复位外设。

参数说明：ulPeripheral，外设号，TimerA0 和 TimerA1 的外设号分别是 0x0E 和 0x0F。

PRCMPeripheralReset()的声明在 prcm.h 中，代码在 prcm.c 中。

③ 配置定时器 TimerConfigure()。配置定时器 TimerConfigure()函数的部分内容如图 4.3 所示。

```
183 void
184 TimerConfigure(unsigned long ulBase, unsigned long ulConfig)
185 {
186
187     ASSERT((ulConfig == TIMER_CFG_ONE_SHOT) ||
188            (ulConfig == TIMER_CFG_ONE_SHOT_UP) ||
189            (ulConfig == TIMER_CFG_PERIODIC) ||
190            (ulConfig == TIMER_CFG_PERIODIC_UP) ||
191            ((ulConfig & 0xff000000) == TIMER_CFG_SPLIT_PAIR));
192     ASSERT(((ulConfig & 0xff000000) != TIMER_CFG_SPLIT_PAIR) ||
193            ((((ulConfig & 0x000000ff) == TIMER_CFG_A_ONE_SHOT) ||
194              ((ulConfig & 0x000000ff) == TIMER_CFG_A_ONE_SHOT_UP) ||
195              ((ulConfig & 0x000000ff) == TIMER_CFG_A_PERIODIC) ||
196              ((ulConfig & 0x000000ff) == TIMER_CFG_A_PERIODIC_UP) ||
197              ((ulConfig & 0x000000ff) == TIMER_CFG_A_CAP_COUNT) ||
198              ((ulConfig & 0x000000ff) == TIMER_CFG_A_CAP_TIME) ||
199              ((ulConfig & 0x000000ff) == TIMER_CFG_A_PWM)) &&
200             (((ulConfig & 0x0000ff00) == TIMER_CFG_B_ONE_SHOT) ||
201              ((ulConfig & 0x0000ff00) == TIMER_CFG_B_ONE_SHOT_UP) ||
202              ((ulConfig & 0x0000ff00) == TIMER_CFG_B_PERIODIC) ||
203              ((ulConfig & 0x0000ff00) == TIMER_CFG_B_PERIODIC_UP) ||
204              ((ulConfig & 0x0000ff00) == TIMER_CFG_B_CAP_COUNT) ||
205              ((ulConfig & 0x0000ff00) == TIMER_CFG_B_CAP_TIME) ||
206              ((ulConfig & 0x0000ff00) == TIMER_CFG_B_PWM))));
```

图 4.3 Timer 项目 TimerConfigure()函数（部分）

```
void TimerConfigure(unsigned long ulBase, unsigned long ulConfig);
```

功能：配置定时器。

参数说明：

- ulBase：定时器基地址，TIMERA0 和 TIMERA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。

- ulConfig：定时器配置，TimerA0 和 TimerA1 的设置值都是 TIMER_CFG_PERIODIC（32 位周期定时器）。ulConfig 包括下列值。
 - ✧ TIMER_CFG_ONE_SHOT：32 位单次定时器，0x0000 0021。
 - ✧ TIMER_CFG_ONE_SHOT_UP：32 位单次向上定时器，0x0000 0031。
 - ✧ TIMER_CFG_PERIODIC：32 位周期定时器，0x0000 0022。
 - ✧ TIMER_CFG_PERIODIC_UP：32 位周期向上定时器，0x0000 0032。
 - ✧ TIMER_CFG_SPLIT_PAIR：双 16 位定时器，0x0400 0000。
 - ✧ TIMER_CFG_A _ONE_SHOT：定时器 A 单次定时器，0x0000 0021。
 - ✧ TIMER_CFG_A _ONE_SHOT_UP：定时器 A 单次向上定时器，0x0000 0031。
 - ✧ TIMER_CFG_A _PERIODIC：定时器 A 周期定时器，0x0000 0022。
 - ✧ TIMER_CFG_A _PERIODIC_UP：定时器 A 周期向上定时器，0x0000 0032。
 - ✧ TIMER_CFG_A_CAP_COUNT：定时器 A 捕捉计数，0x0000 0003。
 - ✧ TIMER_CFG_A_CAP_COUNT_UP：定时器 A 捕捉向上计数，0x0000 0013。
 - ✧ TIMER_CFG_A_CAP_TIME：定时器 A 捕捉计时，0x0000 0007。
 - ✧ TIMER_CFG_A_CAP_TIME_UP：定时器 A 捕捉向上计时，0x0000 0017。
 - ✧ TIMER_CFG_A_PWM：定时器 A PWM 输出，0x0000 000A。
 - ✧ TIMER_CFG_B_ONE_SHOT：定时器 B 单次定时器，0x0000 2100。
 - ✧ TIMER_CFG_B_ONE_SHOT_UP：定时器 B 单次向上定时器，0x0000 3100。
 - ✧ TIMER_CFG_B_PERIODIC：定时器 B 周期定时器，0x0000 2200。
 - ✧ TIMER_CFG_B_PERIODIC_UP：定时器 B 周期向上定时器，0x0000 3200。
 - ✧ TIMER_CFG_B_CAP_COUNT：定时器 B 捕捉计数，0x0000 0300。
 - ✧ TIMER_CFG_B_CAP_COUNT_UP：定时器 B 捕捉向上计数，0x0000 1300。
 - ✧ TIMER_CFG_B_CAP_TIME：定时器 B 捕捉计时，0x0000 0700。
 - ✧ TIMER_CFG_B_CAP_TIME_UP：定时器 B 捕捉向上计时，0x0000 1700。
 - ✧ TIMER_CFG_B_PWM：定时器 B PWM 输出，0x0000 0A00。

TimerConfigure()的声明在 timer.h 中，代码在 timer.c 中。

④ 设置预分频 TimerPrescaleSet()。设置预分频 TimerPrescaleSet()函数的部分内容如图 4.4 所示。

```
358 void
359 TimerPrescaleSet(unsigned long ulBase, unsigned long ulTimer,
360                   unsigned long ulValue)
361 {
362     //
363     // Check the arguments.
364     //
365     ASSERT(TimerBaseValid(ulBase));
366     ASSERT((ulTimer == TIMER_A) || (ulTimer == TIMER_B) ||
367            (ulTimer == TIMER_BOTH));
368     ASSERT(ulValue < 256);
369
370     //
371     // Set the timer A prescaler if requested.
372     //
373     if(ulTimer & TIMER_A)
374     {
375         HWREG(ulBase + TIMER_O_TAPR) = ulValue;
376     }
```

图 4.4　Timer 项目 TimerPrescaleSet()函数（部分）

```
void TimerPrescaleSet(unsigned long ulBase, unsigned long ulTimer,
                      unsigned long ulValue);
```

功能：设置预分频。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA0 和 TimerA1 的设置值都是 TIMER_A（32 位定时器只能使用 TIMER_A）。ulTimer 包括下列值。
 - ✧ TIMER_A：定时器 A，0x0000 00FF。
 - ✧ TIMER_B：定时器 B，0x0000 FF00。
 - ✧ TIMER_BOTH：定时器 A 和定时器 B，0x0000 FFFF。
- ulValue：预分频值，0～255（1～256 分频），TimerA0 和 TimerA1 的设置值都是 0（1 分频即不分频）。

注意：预分频值只对 16 位定时器有效，对 32 位定时器无效。

TimerPrescaleSet()的声明在 timer.h 中，代码在 timer.c 中。

（2）设置定时器中断 Timer_IF_IntSetup()。

```
void Timer_IF_IntSetup(unsigned long ulBase, unsigned long ulTimer,
                       void (*TimerBaseIntHandler)(void));
```

功能：设置定时器中断。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择 TimerA0 和 TimerA1 的设置值都是 TIMER_A（定时器 A）。
- *TimerBaseIntHandler：定时器中断处理程序指针，TimerA0 和 TimerA1 的中断处理程序分别是 TimerBaseIntHandler()和 TimerRefIntHandler()。

Timer_IF_IntSetup()通过调用 TimerIntRegister()和 TimerIntEnable()实现中断处理程序的注册和允许。

① 注册定时器中断 TimerIntRegister()。注册定时器中断 TimerIntRegister()函数的部分内容如图 4.5 所示。

```
755 void
756 TimerIntRegister(unsigned long ulBase, unsigned long ulTimer,
757                  void (*pfnHandler)(void))
758 {
759     //
760     // Check the arguments.
761     //
762     ASSERT(TimerBaseValid(ulBase));
763     ASSERT((ulTimer == TIMER_A) || (ulTimer == TIMER_B) ||
764            (ulTimer == TIMER_BOTH));
765
766     ulBase = ((ulBase == TIMERA0_BASE) ? INT_TIMERA0A :
767               ((ulBase == TIMERA1_BASE) ? INT_TIMERA1A :
768                ((ulBase == TIMERA2_BASE) ? INT_TIMERA2A : INT_TIMERA3A)));
769
```

图 4.5　Timer 项目 TimerIntRegister()函数（部分）

```
void TimerIntRegister(unsigned long ulBase, unsigned long ulTimer,
                      void (*pfnHandler)(void));
```

功能：注册定时器中断。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA0 和 TimerA1 的设置值都是 TIMER_A（定时器 A）。
- *pfnHandler：定时器中断处理程序指针，TimerA0 和 TimerA1 的中断处理程序分别是 TimerBaseIntHandler()和 TimerRefIntHandler()。

TimerIntRegister()的声明在 timer.h 中，代码在 timer.c 中。TimerIntRegister()首先把定时器基地址 ulBase 转换成中断号，然后调用 IntRegister()和 IntEnable()实现中断处理程序的注册和允许。

```
void IntRegister(unsigned long ulInterrupt, void (*pfnHandler)(void));
```

功能：注册中断处理程序。

参数说明：

- ulInterrupt：中断号，TimerA0A 和 TimerA1A 的中断分别是 35 和 37，在 hw_ints.h 中定义。
- *pfnHandler：中断处理程序指针。

IntRegister()的声明在 interrupt.h 中，代码在 interrupt.c 中。

```
void IntEnable(unsigned long ulInterrupt);
```

功能：允许中断。

参数说明：ulInterrupt，中断号，TimerA0A 和 TimerA1A 的中断分别是 35 和 37，在 hw_ints.h 中定义。

IntEnable()的声明在 interrupt.h 中，代码在 interrupt.c 中。

② 允许定时器中断 TimerIntEnable()。允许定时器中断 TimerIntEnable()函数的内容如图 4.6 所示。

```
898 void
899 TimerIntEnable(unsigned long ulBase, unsigned long ulIntFlags)
900 {
901     //
902     // Check the arguments.
903     //
904     ASSERT(TimerBaseValid(ulBase));
905
906     //
907     // Enable the specified interrupts.
908     //
909     HWREG(ulBase + TIMER_O_IMR) |= ulIntFlags;
910 }
```

图 4.6　Timer 项目 TimerIntEnable()函数

```
void TimerIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
```

功能：允许定时器中断。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulIntFlags：定时器中断标志，设置值是 TIMER_TIMA_TIMEOUT（定时器 A 超时中断）。ulIntFlags 包括下列值：
 - ✧ TIMER_TIMA_DMA：定时器 A DMA 中断，0x0000 0020。
 - ✧ TIMER_TIMA_MATCH：定时器 A 匹配中断，0x0000 0010。
 - ✧ TIMER_CAPA_EVENT：定时器 A 捕捉事件中断，0x0000 0004。
 - ✧ TIMER_CAPA_MATCH：定时器 A 捕捉匹配中断，0x0000 0002。
 - ✧ TIMER_TIMA_TIMEOUT：定时器 A 超时中断，0x0000 0001。

TimerIntEnable()的声明在 timer.h 中，代码在 timer.c 中。

（3）启动定时器 Timer_IF_Start()。

```
void Timer_IF_Start(unsigned long ulBase, unsigned long ulTimer,
                    unsigned long ulValue);
```

功能：启动定时器。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA0 和 TimerA1 的设置值都是 TIMER_A（定时器 A）。
- ulValue：定时器初值（毫秒），TimerA0A 和 TimerA1A 的定时器初值分别是 500 ms 和 1 000 ms。

Timer_IF_Start()通过调用 TimerLoadSet()和 TimerEnable()实现定时器的启动。

① 设置定时器初值 TimerLoadSet()。设置定时器初值 TimerLoadSet()函数的部分内容如图 4.7 所示。

```
526 void
527 TimerLoadSet(unsigned long ulBase, unsigned long ulTimer,
528              unsigned long ulValue)
529 {
530     //
531     // Check the arguments.
532     //
533     ASSERT(TimerBaseValid(ulBase));
534     ASSERT((ulTimer == TIMER_A) || (ulTimer == TIMER_B) ||
535            (ulTimer == TIMER_BOTH));
536
537     //
538     // Set the timer A load value if requested.
539     //
540     if(ulTimer & TIMER_A)
541     {
542         HWREG(ulBase + TIMER_O_TAILR) = ulValue;
543     }
```

图 4.7　Timer 项目 TimerLoadSet()函数（部分）

```
void TimerLoadSet(unsigned long ulBase, unsigned long ulTimer,
                  unsigned long ulValue);
```

功能：设置定时器初值。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA0 和 TimerA1 的设置值都是 TIMER_A（定时器 A）。
- ulValue：定时器初值，通过 MILLISECONDS_TO_TICKS()即 80000000/1000×ulValue 将毫秒值转换为时钟值，TimerA0A 和 TimerA1A 的定时器初值分别是 80 000×500 = 40 000 000（周期 0.5 s）和 80 000×1000=80 000 000（周期 1.0 s）。

TimerLoadSet()的声明在 timer.h 中，代码在 timer.c 中。

② 允许定时器 TimerEnable()。允许定时器 TimerEnable()函数的内容如图 4.8 所示。

```
 92 void
 93 TimerEnable(unsigned long ulBase, unsigned long ulTimer)
 94 {
 95     //
 96     // Check the arguments.
 97     //
 98     ASSERT(TimerBaseValid(ulBase));
 99     ASSERT((ulTimer == TIMER_A) || (ulTimer == TIMER_B) ||
100            (ulTimer == TIMER_BOTH));
101
102     //
103     // Enable the timer(s) module.
104     //
105     HWREG(ulBase + TIMER_O_CTL) |= ulTimer & (TIMER_CTL_TAEN | TIMER_CTL_TBEN);
106 }
```

图 4.8　Timer 项目 TimerEnable()函数

```
TimerEnable(unsigned long ulBase, unsigned long ulTimer);
```

功能：允许定时器。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA0 和 TimerA1 的设置值都是 TIMER_A（定时器 A）。

TimerEnable()的声明在 timer.h 中，代码在 timer.c 中。

4．定时器中断处理

TimerA0 和 TimerA1 的中断处理程序 TimerBaseIntHandler()和 TimerRefIntHandler()结构一样，都是首先清除中断标志，然后切换 LED 的显示状态。区别是 TimerA0A 控制绿 LED 的显示，TimerA1A 控制红 LED 的显示。

为了对中断处理程序进行分析，可以在中断处理程序的第一条语句处设置断点。

（1）清除定时器中断标志 Timer_IF_InterruptClear()。

```
void Timer_IF_InterruptClear(unsigned long ulBase);
```

功能：清除定时器中断标志。

参数说明：ulBase，定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。

Timer_IF_InterruptClear()首先调用 TimerIntStatus()获取定时器中断状态，然后调用 TimerIntClear()清除定时器中断标志。

① 获取定时器中断状态 TimerIntStatus()。获取定时器中断状态 TimerIntStatus()函数的内容如图 4.9 所示。

```
959 unsigned long
960 TimerIntStatus(unsigned long ulBase, tBoolean bMasked)
961 {
962     //
963     // Check the arguments.
964     //
965     ASSERT(TimerBaseValid(ulBase));
966
967     //
968     // Return either the interrupt status or the raw interrupt status as
969     // requested.
970     //
971     return(bMasked ? HWREG(ulBase + TIMER_O_MIS) :
972            HWREG(ulBase + TIMER_O_RIS));
973 }
```

图 4.9 Timer 项目 TimerIntStatus()函数

```
unsigned long TimerIntStatus(unsigned long ulBase, tBoolean bMasked);
```

功能：获取定时器中断状态。

参数说明：

- ulBase：定时器基地址 TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- bMasked：中断类型，true 表示屏蔽中断，false 表示原始中断。

返回值：定时器中断状态，值是 1：TIMER_TIMA_TIMEOUT（定时器 A 超时中断）。

TimerIntStatus()的声明在 timer.h 中，代码在 timer.c 中。

② 清除定时器中断标志 TimerIntClear()。清除定时器中断标志 TimerIntClear()函数的内容如图 4.10 所示。

```
1001 void
1002 TimerIntClear(unsigned long ulBase, unsigned long ulIntFlags)
1003 {
1004     //
1005     // Check the arguments.
1006     //
1007     ASSERT(TimerBaseValid(ulBase));
1008
1009     //
1010     // Clear the requested interrupt sources.
1011     //
1012     HWREG(ulBase + TIMER_O_ICR) = ulIntFlags;
1013 }
```

图 4.10 Timer 项目 TimerIntClear()函数

```
void TimerIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

功能：清除定时器中断标志。

参数说明：

- ulBase：定时器基地址，TimerA0 和 TimerA1 的基地址分别是 0x4003 0000 和 0x4003 1000，在 hw_memmap.h 中定义。
- ulIntFlags：中断标志，值是 1，TIMER_TIMA_TIMEOUT（定时器 A 超时中断）。

TimerIntClear()的声明在 timer.h 中，代码在 timer.c 中。

（2）切换 LED GPIO_IF_LedToggle()。

```
void GPIO_IF_LedToggle(unsigned char ucLedNum);
```

功能：切换 LED。

参数说明：ucLedNum，连接 LED 的 GPIO 标识。CC3200 LaunchPad 的 ucLedNum 包括下列值。

- MCU_RED_LED_GPIO：红 LED，0x09（9）。
- MCU_GREEN_LED_GPIO：绿 LED，0x0B（11）。

GPIO_IF_LedToggle()首先调用 GPIO_IF_LedStatus()获取 LED 状态，然后根据 LED 状态调用 GPIO_IF_LedOff()或 GPIO_IF_LedOn()熄灭或点亮 LED。

```
unsigned char GPIO_IF_LedStatus(unsigned char ucGPIONum);
```

功能：获取 LED 状态。

参数说明：ucGPIONum，连接 LED 的 GPIO 标识。

返回值：LED 状态，1 表示点亮，0 表示熄灭。

GPIO_IF_LedStatus()调用 GPIO_IF_Get()获取 GPIO 引脚的值。

```
unsigned char GPIO_IF_Get(unsigned char ucPin, unsigned int uiGPIOPort,
                          unsigned char ucGPIOPin);
```

功能：根据引脚号、端口基地址和引脚位权获取 GPIO 引脚的值。

参数说明：

- ucPin：引脚号，红绿两个 LED 对应的引脚号分别是 9 和 11。
- uiGPIOPort：GPIO 端口基地址，GPIOA1 的端口基地址是 0x4000 5000。
- ucGPIOPin：GPIO 引脚位权，红绿两个 LED 对应的引脚位权分别是 2 和 8。

返回值：GPIO 引脚值，0 或 1。

GPIO_IF_Get()又通过调用 GPIO API GPIOPinRead()读取 GPIO 引脚值。

```
long GPIOPinRead(unsigned long ulPort, unsigned char ucPins);
```

功能：根据端口基地址和引脚位权读取 GPIO 引脚值。

参数说明：

- ulPort：GPIO 端口基地址，GPIOA1 的端口基地址 0x4000 5000。
- ucPins：GPIO 引脚位权，红绿两个 LED 对应的引脚位权分别是 2 和 8。

返回值：GPIO 引脚值。

GPIOPinRead()的声明在 gpio.h 中，代码在 gpio.c 中。

GPIO_IF_LedOff()或 GPIO_IF_LedOn()的使用方法在 LED 闪烁示例程序分析的 LED 操作部分已经介绍，这里不再赘述。

综上所述，Timer 项目使用的 API 函数如表 4.1 所示。

表 4.1　Timer 项目使用的 API 函数

功　能	函数名	调用函数	代码文件
初始化定时器	Timer_IF_Init()	main()	timer_if.c
允许外设时钟	**PRCMPeripheralClkEnable()**	Timer_IF_Init()	prcm.c
复位外设	PRCMPeripheralReset()	Timer_IF_Init()	prcm.c
配置定时器	**TimerConfigure()**	Timer_IF_Init()	timer.c
设置预分频	TimerPrescaleSet()	Timer_IF_Init()	timer.c
设置定时器中断	Timer_IF_IntSetup()	main()	timer_if.c
注册定时器中断	TimerIntRegister()	Timer_IF_IntSetup()	timer.c
允许定时器中断	TimerIntEnable()	Timer_IF_IntSetup()	timer.c
启动定时器	Timer_IF_Start()	main()	timer_if.c
设置定时器初值	**TimerLoadSet()**	Timer_IF_Start()	timer.c
允许定时器	**TimerEnable()**	Timer_IF_Start()	timer.c
清除定时器中断	Timer_IF_InterruptClear()	TimerBaseIntHandler()、TimerRefIntHandler()	timer_if.c
获取中断状态	TimerIntStatus()	Timer_IF_InterruptClear()	timer.c
清除定时器中断	TimerIntClear()	Timer_IF_InterruptClear()	timer.c

4.1.2　PWM 示例程序分析

PWM 示例程序在“example\pwm”目录中，PWM 项目 CC3200 LaunchPad 电路如图 4.11 所示。

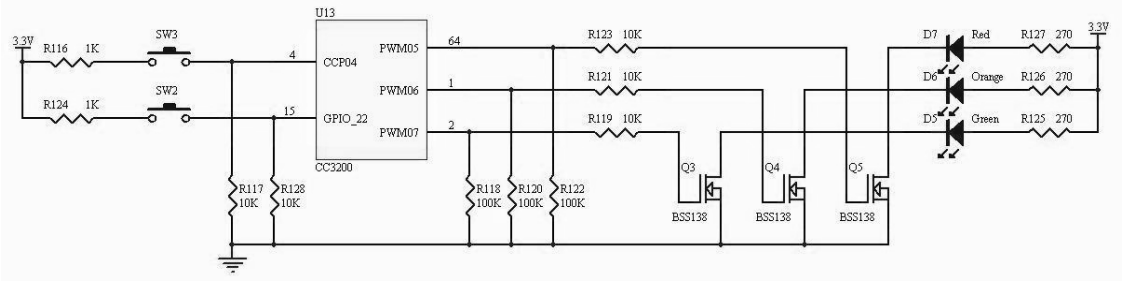


图 4.11　PWM 项目 CC3200 LaunchPad 电路图

CC3200 GPTA2B、GPTA3B 和 GPTA3A 分别通过 PWM05（PIN_64）、PWM06（PIN_01）和 PWM07（PIN_02）输出 PWM 信号（占空比由 0 到 1）控制 3 个 LED 的亮度（由灭逐渐到亮）。

PWM 项目包含下列 3 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 PWM 引脚配置程序文件。
- startup_ccs.c：CCS 启动程序文件。

PWM 项目程序流程图如图 4.12 所示。

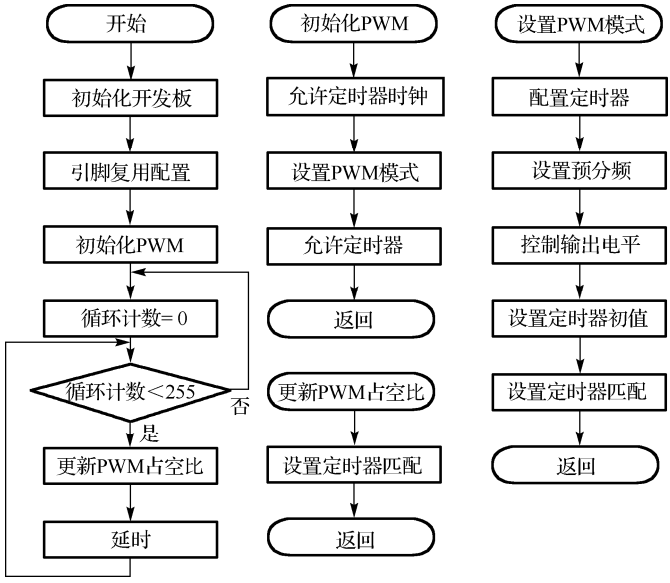


图 4.12　PWM 项目程序流程图

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- PWM 操作。

1．初始化开发板 BoardInit()

初始化开发板包括设置中断向量表、允许处理器中断、允许 SysTick 中断和初始化 MCU 等步骤，调用了相应的中断 API 函数，具体内容和 Blinky 项目相同。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序由 PinMux 应用程序根据设计需要自动生成，程序代码在 pinmux.c 中，其中调用了相应的 PRCM 和 PIN API 函数。

PWM 引脚复用配置包括允许定时器时钟 PRCMPeripheralClkEnable()和配置定时器引脚 PinTypeTimer()。

（1）允许定时器时钟 PRCMPeripheralClkEnable()。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：外设号，TimerA2 和 TimerA3 的外设号分别是 0x10 和 0x11，在 prcm.h 中定义。

- ulClkFlags：时钟标志，设置值是 PRCM_RUN_MODE_CLK（运行模式时钟）。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

（2）配置定时器引脚 PinTypeTimer()。配置定时器引脚 PinTypeTimer()函数的内容如图 4.13 所示。

```
517 void PinTypeTimer(unsigned long ulPin,unsigned long ulPinMode)
518 {
519
520     //
521     // Set the pin to specified mode
522     //
523     PinModeSet(ulPin,ulPinMode);
524
525     //
526     // Set the pin for standard operation
527     //
528     PinConfigSet(ulPin,PIN_STRENGTH_2MA|PIN_STRENGTH_4MA,PIN_TYPE_STD);
529 }
```

图 4.13　PWM 项目 PinTypeTimer()函数

```
void PinTypeTimer(unsigned long ulPin,unsigned long ulPinMode);
```

功能：配置定时器引脚。

参数说明：

- ulPin：引脚号，TimerA2B、TimerA3B 和 TimerA3A 的 PWM 输出引脚号分别是 PIN_64、PIN_01 和 PIN_02。
- ulPinMode：引脚方式，PIN_64、PIN_01 和 PIN_02 的 PIN_MODE_3 为 PWM 方式。

PinTypeTimer()的声明在 pin.h 中，代码在 pin.c 中。PinTypeTimer()通过调用 PinModeSet()和 PinConfigSet()实现引脚类型的配置。

注意：Blinky 项目中的 PinTypeGPIO()和 PWM 项目中的 PinTypeTimer()都对 PIN_64、PIN_01 和 PIN_02 引脚进行配置，但 Blinky 项目配置为 PIN_MODE_0（GPIO 方式），PWM 项目配置为 PIN_MODE_3（PWM 方式）。

3．PWM 操作

PWM 操作主要是初始化 PWM 模块 InitPWMModules()，InitPWMModules()的主要内容是设置 PWM 模式 SetupTimerPWMMode()。

SetupTimerPWMMode()主要包括下列内容，其中最主要的是 TimerMatchSet()。

- 配置定时器：TimerConfigure()。
- 设置预分频：TimerPrescaleSet()。
- 控制输出电平：TimerControlLevel()。
- 设置定时器初值：TimerLoadSet()。
- 设置定时器匹配：TimerMatchSet()。

（1）配置定时器 TimerConfigure()。

```
void TimerConfigure(unsigned long ulBase, unsigned long ulConfig);
```

功能：配置定时器。

参数说明：

- ulBase：定时器基地址，TimerA2 和 TimerA3 的基地址分别是 0x4003 2000 和 0x4003 3000，在 hw_memmap.h 中定义。
- ulConfig：定时器配置，主要包括下列值。
 - ✧ TIMER_CFG_SPLIT_PAIR：双 16 位定时器，0x0400 0000。
 - ✧ TIMER_CFG_A_PWM：定时器 A PWM 输出，0x0000 000A。
 - ✧ TIMER_CFG_B_PWM：定时器 B PWM 输出，0x0000 0A00。

TimerA2 的设置值是 TIMER_CFG_SPLIT_PAIR（双 16 位定时器）和 TIMER_CFG_B_PWM（定时器 B PWM 输出），TimerA3 的设置值是 TIMER_CFG_SPLIT_PAIR（双 16 位定时器）、TIMER_CFG_A_PWM（定时器 A PWM 输出）和 TIMER_CFG_B_PWM（定时器 B PWM 输出）。

TimerConfigure()的声明在 timer.h 中，代码在 timer.c 中。

注意：Timer 项目和 PWM 项目中 TimerConfigure()的主要区别是配置值不同：Timer 项目中配置值是 TIMER_CFG_PERIODIC（32 位周期定时器），PWM 项目中配置值是 TIMER_CFG_ SPLIT_PAIR（双 16 位定时器）和 PWM 输出。

（2）设置预分频 TimerPrescaleSet()。

```
void TimerPrescaleSet(unsigned long ulBase, unsigned long ulTimer,
                      unsigned long ulValue);
```

功能：设置预分频。

参数说明：

- ulBase：定时器基地址，TimerA2 和 TimerA3 的基地址分别是 0x4003 2000 和 0x4003 3000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_B（定时器 B），TimerA3 的设置值是 TIMER_A（定时器 A）和 TIMER_B（定时器 B），包括下列值。
 - ✧ TIMER_A：定时器 A，0x0000 00FF。
 - ✧ TIMER_B：定时器 B，0x0000 FF00。
 - ✧ TIMER_BOTH：定时器 A 和定时器 B，0x0000 FFFF。
- ulValue：预分频值，0～255（1～256 分频），TimerA2 和 TimerA3 的设置值都是 0（1 分频即不分频）。

TimerPrescaleSet()的声明在 timer.h 中，代码在 timer.c 中。

（3）控制输出电平 TimerControlLevel()。

```
void TimerControlLevel(unsigned long ulBase, unsigned long ulTimer,
                       tBoolean bInvert);
```

功能：控制输出电平。

参数说明：

- ulBase：定时器基地址，TimerA2 和 TimerA3 的基地址分别是 0x4003 2000 和 0x4003 3000，在 hw_memmap.h 中定义。

- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_B（定时器 B），TimerA3 的设置值是 TIMER_A（定时器 A）和 TIMER_B（定时器 B）。
- bInvert：定时器输出电平，true 表示低电平，false 表示高电平。

TimerControlLevel()的声明在 timer.h 中，代码在 timer.c 中。

（4）设置定时器初值 TimerLoadSet()。

```
void TimerLoadSet(unsigned long ulBase, unsigned long ulTimer,
                  unsigned long ulValue);
```

功能：设置定时器初值。

参数说明：

- ulBase：定时器基地址，TimerA2 和 TimerA3 的基地址分别是 0x4003 2000 和 0x4003 3000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_B（定时器 B），TimerA3 的设置值是 TIMER_A（定时器 A）和 TIMER_B（定时器 B）。
- ulValue：定时器初值，设置值是 40 035（周期 0.5004735 ms）。定时器初值的计算参见“更新 PWM 占空比”。

TimerLoadSet()的声明在 timer.h 中，代码在 timer.c 中。

（5）设置定时器匹配值 TimerMatchSet()。设置定时器匹配值 TimerMatchSet()函数的部分内容如图 4.14 所示。

```
674 void
675 TimerMatchSet(unsigned long ulBase, unsigned long ulTimer,
676                unsigned long ulValue)
677 {
678     //
679     // Check the arguments.
680     //
681     ASSERT(TimerBaseValid(ulBase));
682     ASSERT((ulTimer == TIMER_A) || (ulTimer == TIMER_B) ||
683            (ulTimer == TIMER_BOTH));
684
685     //
686     // Set the timer A match value if requested.
687     //
688     if(ulTimer & TIMER_A)
689     {
690         HWREG(ulBase + TIMER_O_TAMATCHR) = ulValue;
691     }
```

图 4.14　PWM 项目 TimerMatchSet()函数（部分）

```
void TimerMatchSet(unsigned long ulBase, unsigned long ulTimer,
                   unsigned long ulValue);
```

功能：设置定时器匹配值。

参数说明：

- ulBase：定时器基地址，TimerA2 和 TimerA3 的基地址分别是 0x4003 2000 和 0x4003 3000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_B（定时器 B），TimerA3 的设置值是 TIMER_A（定时器 A）和 TIMER_B（定时器 B），包括下列值。

- ✧ TIMER_A：定时器 A，0x0000 00FF。
- ✧ TIMER_B：定时器 B，0x0000 FF00。
- ✧ TIMER_BOTH：定时器 A 和定时器 B，0x0000 FFFF。

- ulValue：定时器匹配值，0～定时器初值，设置值是 40 035（定时器初值）。

TimerMatchSet()的声明在 timer.h 中，代码在 timer.c 中。

（6）允许定时器 TimerEnable()。

```
TimerEnable(unsigned long ulBase, unsigned long ulTimer);
```

功能：允许定时器。

参数说明：

- ulBase：定时器基地址，TimerA2 和 TimerA3 的基地址分别是 0x4003 2000 和 0x4003 3000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_B（定时器 B），TimerA3 的设置值是 TIMER_A（定时器 A）和 TIMER_B（定时器 B）。

TimerEnable()的声明在 timer.h 中，代码在 timer.c 中。

（7）更新 PWM 占空比 UpdateDutyCycle()。

更新 PWM 占空比通过调用 TimerMatchSet()实现，匹配值是（0～254）×DUTYCYCLE_GRANULARITY（157=40 000/255），实际初值是 255×157=40 035（周期 0.5004735 ms）。

综上所述，PWM 项目使用的 API 函数如表 4.2 所示。

表 4.2　PWM 项目使用的 API 函数

功　能	函数名	调用函数	代码文件
允许外设时钟	**PRCMPeripheralClkEnable()**	PinMuxConfig()、InitPWMModules()	prcm.c
配置定时器引脚	**PinTypeTimer()**	PinMuxConfig()	pin.c
配置定时器	**TimerConfigure()**	SetupTimerPWMMode()	timer.c
设置预分频	TimerPrescaleSet()	SetupTimerPWMMode()	timer.c
控制输出电平	TimerControlLevel()	SetupTimerPWMMode()	timer.c
设置定时器初值	**TimerLoadSet()**	SetupTimerPWMMode()	timer.c
设置定时器匹配值	**TimerMatchSet()**	SetupTimerPWMMode()、UpdateDutyCycle()	timer.c
允许定时器	**TimerEnable()**	InitPWMModules()	timer.c

4.1.3　计数捕捉示例程序分析

计数捕捉示例程序在 example\timer_cc 目录中，Timer_cc 项目 CC3200 LaunchPad 电路如图 4.15 所示。

CC3200 GPTA2A 通过 CCP04 捕捉 PIN_04 输入的信号，测量并显示信号的频率。用按键 SW3 可以产生上升沿和下降沿，程序通过获取定时值计算频率值。

Timer_cc 项目包含下列 4 个程序文件。

- main.c：主程序文件。

- pinmux.c：PinMux 应用程序生成的 UART 和 CCP 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Timer_cc 项目程序流程如图 4.16 所示。

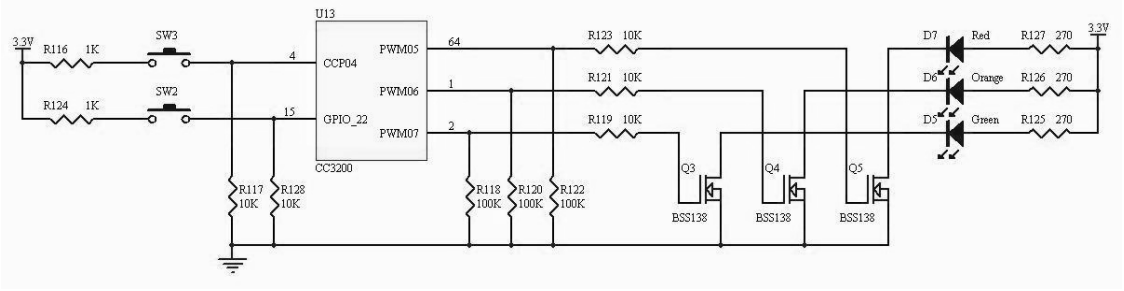


图 4.15　Timer_cc 项目 CC3200 LaunchPad 电路图

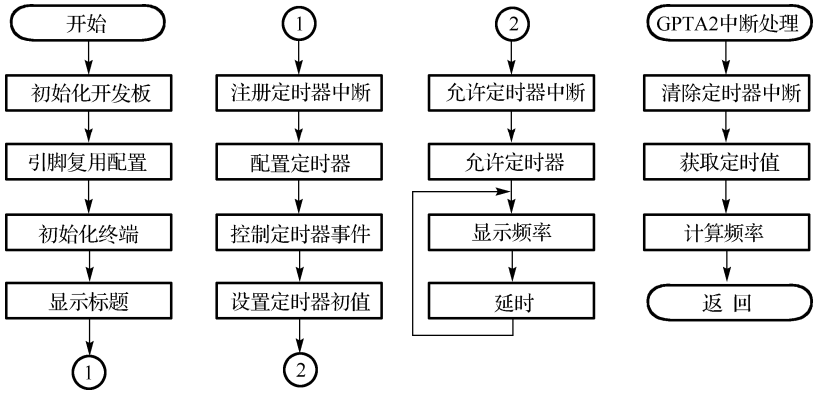


图 4.16　Timer_cc 项目程序流程图

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- 定时器操作。
- 定时器中断处理。

1．初始化开发板 BoardInit()

初始化开发板包括设置中断向量表、允许处理器中断、允许 SysTick 中断和初始化 MCU 等步骤，调用了相应的中断 API 函数，具体内容和 Blinky 项目相同。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序由 PinMux 应用程序根据设计需要自动生成，程序代码在 pinmux.c 中，其中调用了相应的 PRCM 和 PIN API 函数。

Timer_cc 项目引脚复用配置包括允许定时器时钟和配置定时器引脚。

（1）允许定时器时钟 PRCMPeripheralClkEnable()。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：外设号，UARTA0 和 TimerA2 的外设号分别是 0x0C 和 0x10，在 prcm.h 中定义。
- ulClkFlags：时钟标志，设置值是 PRCM_RUN_MODE_CLK（运行模式时钟）。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

（2）配置定时器引脚 PinTypeTimer()。

```
void PinTypeTimer(unsigned long ulPin,unsigned long ulPinMode);
```

功能：配置定时器引脚。

参数说明：

- ulPin：引脚号，TimerA2A 的 CCP 输入引脚号是 PIN_04。
- ulPinMode：引脚方式，PIN_04 的 PIN_MODE_12 为 CCP（捕捉比较 PWM）方式。

PinTypeTimer()的声明在 pin.h 中，代码在 pin.c 中。PinTypeTimer()通过调用 PinModeSet()和 PinConfigSet()实现引脚类型的配置。

注意：PWM 项目和 Timer_cc 项目的定时器引脚都使用 PinTypeTimer()进行配置，但两者的 ulPinMode 参数不同，前者为 PWM 方式，后者为 CCP 方式。

3．定时器操作

定时器操作主要包括下列内容，其中最主要的是 TimerControlEvent()。

- 注册定时器中断：TimerIntRegister()。
- 配置定时器：TimerConfigure()。
- 控制事件类型：TimerControlEvent()。
- 设置定时器初值：TimerLoadSet()。
- 允许定时器中断：TimerIntEnable()。
- 允许定时器：TimerEnable()。

（1）注册定时器中断 TimerIntRegister()。

```
void TimerIntRegister(unsigned long ulBase, unsigned long ulTimer,
                      void (*pfnHandler)(void));
```

功能：注册定时器中断。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_A（定时器 A）。
- *pfnHandler：中断处理程序指针，TimerA2A 的中断处理程序是 TimerIntHandler()。

TimerIntRegister()的声明在 timer.h 中，代码在 timer.c 中。TimerIntRegister()首先把定时器基地址 ulBase 转换成中断号，然后调用 IntRegister()和 IntEnable()实现中断处理程序的注册和允许，具体内容和 Timer 项目相同，这里不再赘述。

（2）配置定时器 TimerConfigure()。

```
void TimerConfigure(unsigned long ulBase, unsigned long ulConfig);
```

功能：配置定时器。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。
- ulConfig：定时器配置，主要包括下列值。
 - ✧ TIMER_CFG_SPLIT_PAIR：双 16 位定时器，0x0400 0000。
 - ✧ TIMER_CFG_A_CAP_COUNT：定时器 A 捕捉计数，0x0000 0003。
 - ✧ TIMER_CFG_A_CAP_TIME：定时器 A 捕捉计时，0x0000 0007。
 - ✧ TIMER_CFG_B_CAP_COUNT：定时器 B 捕捉计数，0x0000 0300。
 - ✧ TIMER_CFG_B_CAP_TIME：定时器 B 捕捉计时，0x0000 0700。

TimerA2 的设置值是 TIMER_CFG_ SPLIT_PAIR（双 16 位定时器）和 TIMER_CFG_A_CAP_TIME（定时器 A 捕捉计时）。TimerConfigure()的声明在 timer.h 中，代码在 timer.c 中。

注意：PWM 项目和 Timer_cc 项目中 TimerConfigure()都是将定时器配置为双 16 位定时器，但 PWM 项目将 16 位定时器配置为 PWM 输出，而 Timer_cc 项目将 16 位定时器配置为捕捉计时。

（3）控制事件类型 TimerControlEvent()。控制事件类型 TimerControlEvent()函数的内容如图 4.17 所示。

```
284 void
285 TimerControlEvent(unsigned long ulBase, unsigned long ulTimer,
286                    unsigned long ulEvent)
287 {
288     //
289     // Check the arguments.
290     //
291     ASSERT(TimerBaseValid(ulBase));
292     ASSERT((ulTimer == TIMER_A) || (ulTimer == TIMER_B) ||
293            (ulTimer == TIMER_BOTH));
294
295     //
296     // Set the event type.
297     //
298     ulEvent &= ulTimer & (TIMER_CTL_TAEVENT_M | TIMER_CTL_TBEVENT_M);
299     HWREG(ulBase + TIMER_O_CTL) = ((HWREG(ulBase + TIMER_O_CTL) &
300                                     ~(TIMER_CTL_TAEVENT_M |
301                                       TIMER_CTL_TBEVENT_M)) | ulEvent);
302 }
```

图 4.17　Timer_cc 项目 TimerControlEvent()函数

```
void TimerControlEvent(unsigned long ulBase, unsigned long ulTimer,
                       unsigned long ulEvent);
```

功能：控制事件类型。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_A（定时器 A）。
- ulEvent：事件类型，TimerA2 的设置值是 TIMER_EVENT_POS_EDGE（上升沿计数）。ulEvent 包括下列值。
 - ✧ TIMER_EVENT_POS_EDGE：上升沿计数，0x0000 0000。
 - ✧ TIMER_EVENT_NEG_EDGE：下降沿计数，0x0000 0404。
 - ✧ TIMER_EVENT_BOTH_EDGES：上升沿和下降沿计数，0x0000 0C0C。

TimerControlEvent()的声明在 timer.h 中，代码在 timer.c 中。

（4）设置定时器初值 TimerLoadSet()。

```
void TimerLoadSet(unsigned long ulBase, unsigned long ulTimer,
                  unsigned long ulValue);
```

功能：设置定时器初值。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_A（定时器 A）。
- ulValue：定时器初值，TimerA2A 的定时器初值是 0xFFFF（最大值）。

TimerLoadSet()的声明在 timer.h 中，代码在 timer.c 中。

（5）允许定时器中断 TimerIntEnable()。

```
void TimerIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
```

功能：允许定时器中断。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。
- ulIntFlags：定时器中断标志，设置值是 TIMER_ CAPA_EVENT（定时器 A 捕捉事件中断）。ulIntFlags 包括下列值。
 - ✧ TIMER_TIMA_DMA：定时器 A DMA 中断，0x0000 0020。
 - ✧ TIMER_TIMA_MATCH：定时器 A 匹配中断，0x0000 0010。
 - ✧ TIMER_CAPA_EVENT：定时器 A 捕捉事件中断，0x0000 0004。
 - ✧ TIMER_CAPA_MATCH：定时器 A 捕捉匹配中断，0x0000 0002。
 - ✧ TIMER_TIMA_TIMEOUT：定时器 A 超时中断，0x0000 0001。

TimerIntEnable()的声明在 timer.h 中，代码在 timer.c 中。

（6）允许定时器 TimerEnable()。

```
TimerEnable(unsigned long ulBase, unsigned long ulTimer);
```

功能：允许定时器。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。

- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_A（定时器 A）。

TimerEnable()的声明在 timer.h 中，代码在 timer.c 中。

4．定时器中断处理

TimerA2A 的中断处理程序 TimerIntHandler()首先清除中断标志，然后获取定时值，并根据相邻的两个定时值计算信号频率。

为了对中断处理程序进行分析，可以在中断处理程序的第一条语句处设置断点。为了产生定时器捕捉事件，可以按开发板上的 SW3 按键，每按一次，频率改变一次。

（1）清除定时器中断标志 TimerIntClear()。

```
void TimerIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

功能：清除定时器中断标志。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。
- ulIntFlags：中断标志，值是 4：TIMER_ CAPA_EVENT（定时器 A 捕捉事件中断）。

TimerIntClear()的声明在 timer.h 中，代码在 timer.c 中。

（2）获取定时值 TimerValueGet()。获取定时值 TimerValueGet()函数的内容如图 4.18 所示。

```
602 unsigned long
603 TimerValueGet(unsigned long ulBase, unsigned long ulTimer)
604 {
605     //
606     // Check the arguments.
607     //
608     ASSERT(TimerBaseValid(ulBase));
609     ASSERT((ulTimer == TIMER_A) || (ulTimer == TIMER_B));
610
611     //
612     // Return the appropriate timer value.
613     //
614     return((ulTimer == TIMER_A) ? HWREG(ulBase + TIMER_O_TAR) :
615            HWREG(ulBase + TIMER_O_TBR));
616 }
```

图 4.18　Timer_cc 项目 TimerValueGet()函数

```
unsigned long TimerValueGet(unsigned long ulBase, unsigned long ulTimer);
```

功能：获取定时值。

参数说明：

- ulBase：定时器基地址，TimerA2 的基地址是 0x4003 2000，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，TimerA2 的设置值是 TIMER_A（定时器 A）。

返回值：定时器当前值。

TimerValueGet()的声明在 timer.h 中，代码在 timer.c 中。

综上所述，Timer_cc 项目使用的 API 函数如表 4.3 所示。

表 4.3　Timer_cc 项目使用的 API 函数

功　能	函数名	调用函数	代码文件
允许外设时钟	**PRCMPeripheralClkEnable()**	PinMuxConfig()	prcm.c
配置定时器引脚	**PinTypeTimer()**	PinMuxConfig()	pin.c
注册定时器中断	TimerIntRegister()	main()	timer.c
配置定时器	**TimerConfigure()**	main()	timer.c
设置控制事件	**TimerControlEvent()**	main()	timer.c
设置定时器初值	**TimerLoadSet()**	main()	timer.c
允许定时器中断	TimerIntEnable()	main()	timer.c
允许定时器	**TimerEnable()**	main()	timer.c
清除定时器中断	TimerIntClear()	TimerIntHandler()	timer.c
获取定时值	**TimerValueGet()**	TimerIntHandler()	timer.c

4.1.4　GPT 使用简介

GPT 的使用主要包括引脚复用配置、初始化定时器、设置定时器中断、启动定时器、清除定时器中断标志、PWM 操作和计数捕捉操作等。

1．引脚复用配置

引脚复用配置包括允许定时器时钟 PRCMPeripheralClkEnable()和配置定时器引脚 PinTypeTimer()（配置 PWM 输出引脚和计数捕捉输入引脚），API 声明如下。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
void PinTypeTimer(unsigned long ulPin,unsigned long ulPinMode);
```

2．初始化定时器

初始化定时器 Timer_IF_Init()包括允许定时器时钟 PRCMPeripheralClkEnable()、复位定时器 PRCMPeripheralReset()、配置定时器 TimerConfigure()和设置预分频 TimerPrescale Set()，API 声明如下。

```
void Timer_IF_Init(unsigned long ePeripheral, unsigned long ulBase,
                   unsigned long ulConfig, unsigned long ulTimer,
                   unsigned long ulValue);
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
void PRCMPeripheralReset(unsigned long ulPeripheral);
void TimerConfigure(unsigned long ulBase, unsigned long ulConfig);
void TimerPrescaleSet(unsigned long ulBase, unsigned long ulTimer,
                      unsigned long ulValue);
```

Timer_IF_Init()和 TimerConfigure()的 ulConfig 参数中包含整体参数和局部参数，整体参数用于 32 位或双 16 位配置，局部参数用于 TIMER_A（定时器 A）和 TIMER_B（定时器 B）配置。

当定时器配置为 32 位时（参数中无 TIMER_CFG_SPLIT_PAIR 项），局部参数只包括

TIMER_A（定时器 A）配置参数，TIMER_B（定时器 B）配置参数无效，且 TIMER_A（定时器 A）配置参数只能使用下列值。

- TIMER_CFG_ONE_SHOT：32 位单次定时器，0x0000 0021。
- TIMER_CFG_ONE_SHOT_UP：32 位单次向上定时器，0x0000 0031。
- TIMER_CFG_PERIODIC：32 位周期定时器，0x0000 0022。
- TIMER_CFG_PERIODIC_UP：32 位周期向上定时器，0x0000 0032。

当定时器配置为双 16 位时（参数中有 TIMER_CFG_SPLIT_PAIR 项），局部参数必须同时包括 TIMER_A（定时器 A）和 TIMER_B（定时器 B）配置参数，不能单独配置。

TIMER_A（定时器 A）和 TIMER_B（定时器 B）配置参数包括下列值。

- TIMER_CFG_SPLIT_PAIR：双 16 位定时器，0x0400 0000。
- TIMER_CFG_A _ONE_SHOT：定时器 A 单次定时器，0x0000 0021。
- TIMER_CFG_A _ONE_SHOT_UP：定时器 A 单次向上定时器，0x0000 0031。
- TIMER_CFG_A _PERIODIC：定时器 A 周期定时器，0x0000 0022。
- TIMER_CFG_A _PERIODIC_UP：定时器 A 周期向上定时器，0x0000 0032。
- TIMER_CFG_A_CAP_COUNT：定时器 A 捕捉计数，0x0000 0003。
- TIMER_CFG_A_CAP_COUNT_UP：定时器 A 捕捉向上计数，0x0000 0013。
- TIMER_CFG_A_CAP_TIME：定时器 A 捕捉计时，0x0000 0007。
- TIMER_CFG_A_CAP_TIME_UP 定时器 A 捕捉向上计时，0x0000 0017。
- TIMER_CFG_A_PWM：定时器 A PWM 输出，0x0000 000A。
- TIMER_CFG_B_ONE_SHOT：定时器 B 单次定时器，0x0000 2100。
- TIMER_CFG_B_ONE_SHOT_UP：定时器 B 单次向上定时器，0x0000 3100。
- TIMER_CFG_B_PERIODIC：定时器 B 周期定时器，0x0000 2200。
- TIMER_CFG_B_PERIODIC_UP：定时器 B 周期向上定时器，0x0000 3200。
- TIMER_CFG_B_CAP_COUNT：定时器 B 捕捉计数，0x0000 0300。
- TIMER_CFG_B_CAP_COUNT_UP：定时器 B 捕捉向上计数，0x0000 1300。
- TIMER_CFG_B_CAP_TIME：定时器 B 捕捉计时，0x0000 0700。
- TIMER_CFG_B_CAP_TIME_UP：定时器 B 捕捉向上计时，0x0000 1700。
- TIMER_CFG_B_PWM：定时器 B PWM 输出，0x0000 0A00。

Timer_IF_Init()的 ulTimer 参数只能用于预分频值 ulValue 的设置，包括下列值。

- TIMER_A：定时器 A，0x0000 00FF。
- TIMER_B：定时器 B，0x0000 FF00。
- TIMER_BOTH：定时器 A 和定时器 B，0x0000 FFFF。

预分频值可以单独设置（ulTimer 取值 TIMER_A 或 TIMER_B），也可以同时设置（ulTimer 取值 TIMER_BOTH）。

3．设置定时器中断

设置定时器中断 Timer_IF_IntSetup()包括注册定时器中断 TimerIntRegister()和允许定时器中断 TimerIntEnable()，API 声明如下。

```
void Timer_IF_IntSetup(unsigned long ulBase, unsigned long ulTimer,
                       void (*TimerBaseIntHandler)(void));
void TimerIntRegister(unsigned long ulBase, unsigned long ulTimer,
                      void (*pfnHandler)(void));
void TimerIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
```

TimerIntEnable()的 ulIntFlags 参数包括下列值。

- TIMER_TIMB_DMA：定时器 B DMA 中断，0x0000 2000。
- TIMER_TIMB_MATCH：定时器 B 匹配中断，0x0000 0800。
- TIMER_CAPB_EVENT：定时器 B 捕捉事件中断，0x0000 0400。
- TIMER_CAPB_MATCH：定时器 B 捕捉匹配中断，0x0000 0200。
- TIMER_TIMB_TIMEOUT：定时器 B 超时中断，0x0000 0100。
- TIMER_TIMA_DMA：定时器 A DMA 中断，0x0000 0020。
- TIMER_TIMA_MATCH：定时器 A 匹配中断，0x0000 0010。
- TIMER_CAPA_EVENT：定时器 A 捕捉事件中断，0x0000 0004。
- TIMER_CAPA_MATCH：定时器 A 捕捉匹配中断，0x0000 0002。
- TIMER_TIMA_TIMEOUT：定时器 A 超时中断，0x0000 0001。

Timer_IF_IntSetup()在函数内部将中断类型固定设置为 TIMER_TIMA_TIMEOUT（定时器 A 超时中断）和 TIMER_TIMB_TIMEOUT（定时器 B 超时中断），因此设置其他中断时不能使用 Timer_IF_IntSetup()。

4．启动定时器

启动定时器 Timer_IF_Start()包括设置定时器初值 TimerLoadSet()和允许定时器 Timer Enable()，API 声明如下。

```
void Timer_IF_Start(unsigned long ulBase, unsigned long ulTimer,
                    unsigned long ulValue);
void TimerLoadSet(unsigned long ulBase, unsigned long ulTimer,
                  unsigned long ulValue);
TimerEnable(unsigned long ulBase, unsigned long ulTimer);
```

5．清除定时器中断标志

清除定时器中断标志 Timer_IF_InterruptClear()包括获取定时器中断状态 TimerInt Status()和清除定时器中断标志 TimerIntClear()，API 声明如下。

```
void Timer_IF_InterruptClear(unsigned long ulBase);
unsigned long TimerIntStatus(unsigned long ulBase, tBoolean bMasked);
void TimerIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

6．PWM 操作和计数捕捉操作

PWM 操作和计数捕捉操作除进行基本的定时器操作外，PWM 操作主要是控制输出电平 TimerControlLevel()和设置定时器匹配 TimerMatchSet()，计数捕捉操作主要是控制事件类型 TimerControlEvent()和获取定时值 TimerValueGet()，API 声明如下。

```
void TimerControlLevel(unsigned long ulBase, unsigned long ulTimer,
                       tBoolean bInvert);
void TimerMatchSet(unsigned long ulBase, unsigned long ulTimer,
                   unsigned long ulValue);
void TimerControlEvent(unsigned long ulBase, unsigned long ulTimer,
                       unsigned long ulEvent);
unsigned long TimerValueGet(unsigned long ulBase, unsigned long ulTimer);
```

TimerControlEvent()的参数 ulEvent 包括下列值。

- TIMER_EVENT_POS_EDGE：上升沿计数，0x0000 0000。
- TIMER_EVENT_NEG_EDGE：下降沿计数，0x0000 0404。
- TIMER_EVENT_BOTH_EDGES：上升沿和下降沿计数，0x0000 0C0C。

4.1.5 矩形波周期和脉宽测量程序设计

下面将定时器的 PWM 输出和计数捕捉输入结合起来，设计矩形波周期和脉宽的测量程序。具体内容是：GPTA2B、GPTA3B 和 GPTA3A 设置为 PWM 模式，周期依次为 150 ms、100 ms 和 50 ms，脉宽依次为 75 ms、50 ms 和 25 ms，分别通过 PWM05（PIN_64，P3.9）、PWM06（PIN_01，P1.9）和 PWM07（PIN_02，P1.10）输出矩形波，控制 3 个 LED 的亮度。

GPTA2A 设置为计数捕捉模式，通过 CCP04（PIN_04，P1.3）捕捉输入信号，测量信号的周期和脉宽。

具体方法是在 Timer_cc 项目的基础上做如下修改。

（1）将 pinmux.c 文件的 PinMuxConfig()函数体修改为下列语句。

```
//Enable Peripheral Clocks                          允许外设时钟
MAP_PRCMPeripheralClkEnable(PRCM_UARTA0, PRCM_RUN_MODE_CLK);
MAP_PRCMPeripheralClkEnable(PRCM_TIMERA2, PRCM_RUN_MODE_CLK);
MAP_PRCMPeripheralClkEnable(PRCM_TIMERA3, PRCM_RUN_MODE_CLK);

//Configure PIN_55 for UART0 UART0_TX          配置 PIN_55 作为 UART0_TX
MAP_PinTypeUART(PIN_55, PIN_MODE_3);
//Configure PIN_57 for UART0 UART0_RX          配置 PIN_57 作为 UART0_RX
MAP_PinTypeUART(PIN_57, PIN_MODE_3);

//Configure PIN_64 for TIMERPWM5 GT_PWM05      配置 PIN_64 作为 GT_PWM05
MAP_PinTypeTimer(PIN_64, PIN_MODE_3);
//Configure PIN_01 for TIMERPWM6 GT_PWM06      配置 PIN_01 作为 GT_PWM06
MAP_PinTypeTimer(PIN_01, PIN_MODE_3);
//Configure PIN_02 for TIMERPWM7 GT_PWM07      配置 PIN_02 作为 GT_PWM07
MAP_PinTypeTimer(PIN_02, PIN_MODE_3);

//Configure PIN_04 for TIMERCP4 GT_CCP04       配置 PIN_04 作为 GT_CCP04
MAP_PinTypeTimer(PIN_04, PIN_MODE_12);
```

（2）将 main.c 中的语句

```
static unsigned long g_ulSamples[2];
```

修改为

```
static unsigned long g_ulSamples[3];
```

（3）将 main()函数体修改为下列语句。

```
//Initialize board configurations          初始化开发板
BoardInit();

//Pinmux for UART and TIMER                UART 和 TIMER 引脚复用配置
PinMuxConfig();

//Configuring UART                         配置 UART
InitTerm();

//Display Application Banner               显示标题
DisplayBanner(APP_NAME);

//Registers an interrupt handler for the timer interrupt  注册定时器中断
MAP_TimerIntRegister(TIMERA2_BASE, TIMER_A, TimerIntHandler);
//Enables timer interrupt                  允许定时器中断：定时器 A 捕捉事件
MAP_TimerIntEnable(TIMERA2_BASE, TIMER_CAPA_EVENT);
//Configures the timer(s)                  配置定时器
MAP_TimerConfigure(TIMERA2_BASE,
    TIMER_CFG_SPLIT_PAIR|TIMER_CFG_A_CAP_TIME|TIMER_CFG_B_PWM);
MAP_TimerConfigure(TIMERA3_BASE,
    TIMER_CFG_SPLIT_PAIR|TIMER_CFG_A_PWM|TIMER_CFG_B_PWM);
//Set the timer prescale value，设置定时器预分频：80 MHz/200=400 kHz
MAP_TimerPrescaleSet(TIMERA2_BASE, TIMER_BOTH, 199);
MAP_TimerPrescaleSet(TIMERA3_BASE, TIMER_BOTH, 199);

//Controls the event type                  控制事件类型：双沿
MAP_TimerControlEvent(TIMERA2_BASE, TIMER_A, TIMER_EVENT_BOTH_EDGES);

//Sets the timer load value                设置定时器初值
MAP_TimerLoadSet(TIMERA2_BASE, TIMER_A, 65535);
MAP_TimerLoadSet(TIMERA2_BASE, TIMER_B, 60000);   //60000/400=150 ms
MAP_TimerLoadSet(TIMERA3_BASE, TIMER_A, 40000);   //40000/400=100 ms
MAP_TimerLoadSet(TIMERA3_BASE, TIMER_B, 20000);   //20000/400=50 ms

//Sets the timer match value               设置定时器匹配值
MAP_TimerMatchSet(TIMERA2_BASE, TIMER_B, 30000);  //30000/400=75 ms
MAP_TimerMatchSet(TIMERA3_BASE, TIMER_A, 20000);  //20000/400=50 ms
MAP_TimerMatchSet(TIMERA3_BASE, TIMER_B, 10000);  //10000/400=25 ms

//Enables the timer(s)                     允许定时器
MAP_TimerEnable(TIMERA2_BASE, TIMER_BOTH);
MAP_TimerEnable(TIMERA3_BASE, TIMER_BOTH);
```

```
while(1)
{
    if(g_ulFreq >= 3)
    {
        MAP_TimerDisable(TIMERA2_BASE, TIMER_A);    //禁止定时器
        Report("Period: %03dms  ",
                        (65536 + g_ulSamples[0] - g_ulSamples[2])/400);
        Report("Width: %03dms\r\n", (g_ulSamples[0] - g_ulSamples[1])/400);
        g_ulFreq = 0;
        MAP_UtilsDelay(80000000/5);
        MAP_TimerEnable(TIMERA2_BASE, TIMER_A);     //允许定时器
    }
}
```

（4）将 TimerIntHandler()中断处理程序修改为下列语句。

```
//Clear timer interrupt                    清除定时器中断：定时器 A 捕捉事件
MAP_TimerIntClear(TIMERA2_BASE, TIMER_CAPA_EVENT);

//Gets timer value                         获取定时值
g_ulSamples[g_ulFreq++] = MAP_TimerValueGet(TIMERA2_BASE, TIMER_A);
```

下载并运行程序，依次用导线连接 P1.3 和 P3.9、P1.3 和 P1.9 以及 P1.3 和 P1.10，在 PC 超级终端窗口将依次显示下列内容。

```
Period: 150ms  Width: 075 ms
Period: 100ms  Width: 050 ms
Period: 050ms  Width: 025 ms
```

4.2 ADC 应用

模/数转换器 ADC 的主要功能是将模拟信号转化为数字信号，以便微控制器进行数据处理。ADC 按转换原理可以分为逐次比较型、双积分型和Σ-Δ型。

逐次比较型 ADC 通过逐次比较将模拟信号转化为数字信号，转换速度快，但精度较低，是最常用的 ADC。

双积分型 ADC 通过两次积分将模拟信号转化为数字信号，精度高，抗干扰能力强，但速度较慢，主要用于万用表等测量仪器。

Σ-Δ型 ADC 具有逐次比较型和双积分型的双重优点，正在逐步广泛地得到应用。

CC3200 ADC 是逐次比较型 ADC，具有下列特性。

- 8 个通道，每个通道各有 4 个 32 位的 FIFO 用于存放时间标记和转换数据。
 - ✧ 4 个外部模拟输入通道供用户使用（ADC_CH0～3，最大输入电压 1.4 V）。
 - ✧ 4 个内部模拟通道用于网络子系统。
- 12 位分辨率，满量程电压为 1.4 V，最小测量电压为 0.34 mV（1.4 V/4096）。
- 自动循环采样，每个通道固定采样间隔 16 μs（采样率 62.5 kSps）。

- 自动为每个采样添加 17 位基于系统时钟（40 MHz）的时间标记。
- 专用 DMA 通道用于传送 ADC 数据到应用程序 RAM。

4.2.1　ADC 示例程序分析

ADC 示例程序在“example\adc”目录中，CC3200 ADC 的 4 个外部模拟输入引脚分别是 PIN_57、PIN_58、PIN_59 和 PIN_60，ADC 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- adc_userinput.c：ADC 用户输入程序文件。
- uart_if.c：UART 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

ADC 项目程序流程如图 4.19 所示。

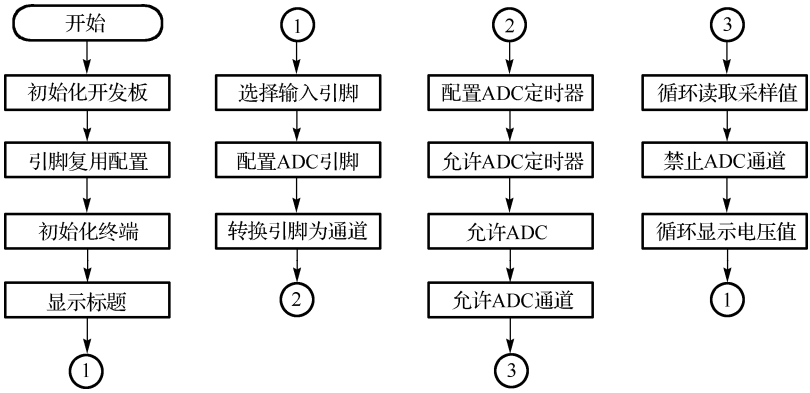


图 4.19　ADC 项目程序流程图

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化开发板：BoardInit()。
- 引脚复用配置：PinMuxConfig()。
- 终端操作。
- ADC 操作。

1．初始化开发板 BoardInit()

初始化开发板包括设置中断向量表、允许处理器中断、允许 SysTick 中断和初始化 MCU 等步骤，调用了相应的中断 API 函数，具体内容和 Blinky 项目相同。

2．引脚复用配置 PinMuxConfig()

引脚复用配置程序由 PinMux 应用程序根据设计需要自动生成，程序代码在 pinmux.c 中，其中调用了相应的 PRCM 和 PIN API 函数。

ADC 项目的引脚复用配置和 Uart_demo 项目相同，主要包括允许 UART 时钟和配置 UART 引脚。

3．终端操作

终端操作包括初始化终端、显示标题、选择输入引脚和显示电压值等。

（1）初始化终端 InitTerm()的使用方法与 Uart_demo 项目相同。

（2）显示标题 DisplayBanner()通过调用 Report()显示下列标题。

```
*********************************************************************
                        CC3200 ADC Reference Application
*********************************************************************
```

（3）选择输入引脚通过调用 ReadFromUser()实现，包括显示用法、显示提示符、获取命令和分析命令等，程序代码在 adc_userinput.c 中。

① 显示用法 UsageDisplay()通过调用 UART_PRINT()即 Report()显示下列信息。

```
Command Usage                                          命令用法
---------------------------------------------------------------------------
adcdemo <adc_inputpin>

Parameters                                             参数
---------------------------------------------------------------------------
adc_inputpin - Pin to which Analog signal is connected  [58|59|60]
                              adc_inputpin – 模拟信号连接的输入引脚：58、59 或 60
---------------------------------------------------------------------------
```

② 显示提示符通过调用 UART_PRINT()即 Report()显示提示符“cmd#”。

③ 获取命令 GetCmd()通过终端获取并回显用户输入的命令，例如输入下列命令。

```
adcdemo 58
```

④ 分析命令通过调用 ADC 分析函数 ADCParser()将模拟信号连接的输入引脚 58、59 或 60 转换为对应的引脚名称 PIN_58（0x39）、PIN_59（0x3A）或 PIN_60（0x3B）。

（4）显示电压值通过调用 UART_PRINT()即 Report()显示 128 个采样值（bits[13:2]）对应的电压值。

4．ADC 操作

ADC 操作主要包括下列内容。

- 配置 ADC 引脚：PinTypeADC()。
- 配置 ADC 定时器：ADCTimerConfig()。
- 允许 ADC 定时器：ADCTimerEnable()。
- 允许 ADC：ADCEnable()。
- 允许 ADC 通道：ADCChannelEnable()。
- 读取采样值：ADCFIFOLvlGet()。
- 禁止 ADC 通道：ADCChannelDisable()。

（1）配置 ADC 引脚 PinTypeADC()。配置 ADC 引脚 PinTypeADC()函数的内容如图 4.20 所示。

```
622 void PinTypeADC(unsigned long ulPin,unsigned long ulPinMode)
623 {
624   //
625   // Configure the Pin
626   //
627   PinConfigSet(ulPin,PIN_STRENGTH_2MA,PIN_TYPE_ANALOG);
628 }
```

图 4.20　ADC 项目 PinTypeADC()函数

```
void PinTypeADC(unsigned long ulPin, unsigned long ulPinMode);
```

功能：配置 ADC 引脚。

参数说明：

- ulPin：引脚号，ADC 的输入引脚号是 PIN_57、PIN_58、PIN_59 或 PIN_60。
- ulPinMode：引脚方式，设置值是 PIN_MODE_255。ADC 的引脚方式参数没用，而是使用 PIN_TYPE_ ANALOG（模拟引脚）调用 PinConfigSet()。

PinTypeADC()的声明在 pin.h 中，代码在 pin.c 中。PinTypeADC()通过调用 PinConfigSet()将 ADC 输入引脚类型设置为 PIN_TYPE_ANALOG（模拟引脚）。

（2）配置 ADC 定时器 ADCTimerConfig()。

```
void ADCTimerConfig(unsigned long ulBase, unsigned long ulValue);
```

功能：配置 ADC 定时器。

参数说明：

- ulBase：ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。
- ulValue：配置值，0～2^{17}（应为 1<<17，2^17 的结果是 19，^是异或运算符）。

ADCTimerConfig()的声明在 adc.h 中，代码在 adc.c 中。

（3）允许 ADC 定时器 ADCTimerEnable()。

```
void ADCTimerEnable(unsigned long ulBase);
```

功能：允许 ADC 定时器。

参数说明：ulBase，ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。

ADCTimerEnable()的声明在 adc.h 中，代码在 adc.c 中。

（4）允许 ADC 函数 ADCEnable()。

```
void ADCEnable(unsigned long ulBase);
```

功能：允许 ADC。

参数说明：ulBase，ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。

ADCEnable()的声明在 adc.h 中，代码在 adc.c 中。

（5）允许 ADC 通道 ADCChannelEnable()。允许 ADC 通道 ADCChannelEnable()函数的内容如图 4.21 所示。

```
106 void ADCChannelEnable(unsigned long ulBase, unsigned long ulChannel)
107 {
108   unsigned long ulCh;
109
110   ulCh =  (ulChannel == ADC_CH_0)? 0x02 :
111           (ulChannel == ADC_CH_1)? 0x04 :
112           (ulChannel == ADC_CH_2)? 0x08 : 0x10;
113
114   HWREG(ulBase + ADC_O_ADC_CH_ENABLE) |= ulCh;
115 }
```

图 4.21　ADC 项目 ADCChannelEnable()函数

```
void ADCChannelEnable(unsigned long ulBase, unsigned long ulChannel);
```

功能：允许 ADC 通道。

参数说明：

- ulBase：ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。
- ulChannel：ADC 通道，取值为 ADC_CH_0～3（ADC 通道 0～3）。

ADCChannelEnable()的声明在 adc.h 中，代码在 adc.c 中。

（6）读取采样值。读取采样值通过读取 ADC FIFO 数据实现，在读取 ADC FIFO 数据前需要获取 ADC FIFO 阈值，根据 FIFO 阈值判断 FIFO 中是否有数据可以读取。

① 获取 ADC FIFO 阈值 ADCFIFOLvlGet()。获取 ADC FIFO 阈值 ADCFIFOLvlGet()函数的内容如图 4.22 所示。

```
636 unsigned char ADCFIFOLvlGet(unsigned long ulBase, unsigned long ulChannel)
637 {
638   unsigned long ulOffset;
639
640   //
641   // Get the fifo level register offset for specified channel
642   //
643   ulOffset = ADC_O_adc_ch0_fifo_lvl + ulChannel;
644
645   //
646   // Return FIFO level
647   //
648   return(HWREG(ulBase + ulOffset) & 0x7);
649 }
```

图 4.22　ADC 项目 ADCFIFOLvlGet()函数

```
unsigned char ADCFIFOLvlGet(unsigned long ulBase, unsigned long ulChannel);
```

功能：获取 ADC FIFO 阈值。

参数说明：

- ulBase：ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。
- ulChannel：ADC 通道，取值为 ADC_CH_0～3。

返回值：FIFO 阈值，0～3。

ADCFIFOLvlGet()的声明在 adc.h 中，代码在 adc.c 中。

② 读取 ADC FIFO 数据 ADCFIFORead()。读取 ADC FIFO 数据 ADCFIFORead()函数的内容如图 4.23 所示。

```
671 unsigned long ADCFIFORead(unsigned long ulBase, unsigned long ulChannel)
672 {
673   unsigned long ulOffset;
674
675   //
676   // Get the fifo register offset for specified channel
677   //
678   ulOffset = ADC_O_channel0FIFODATA + ulChannel;
679
680   //
681   // Return FIFO level
682   //
683   return(HWREG(ulBase + ulOffset));
684 }
```

图 4.23　ADC 项目 ADCFIFORead()函数

```
unsigned long ADCFIFORead(unsigned long ulBase, unsigned long ulChannel);
```

功能：读取 ADC FIFO 数据。

参数说明：

- ulBase：ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。
- ulChannel：ADC 通道，取值为 ADC_CH_0～3。

返回值：采样值，包括时间标记。

ADCFIFORead()的声明在 adc.h 中，代码在 adc.c 中。

（7）禁止 ADC 通道 ADCChannelDisable()。

```
void ADCChannelDisable(unsigned long ulBase, unsigned long ulChannel);
```

功能：禁止 ADC 通道。

参数说明：

- ulBase：ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。
- ulChannel：ADC 通道，取值为 ADC_CH_0～3（ADC 通道 0～3）。

ADCChannelDisable()的声明在 adc.h 中，代码在 adc.c 中。

综上所述，ADC 项目使用的 API 函数如表 4.4 所示。

表 4.4　ADC 项目使用的 API 函数

功　能	函数名	调用函数	代码文件
配置 ADC 引脚	**PinTypeADC()**	main()	pin.c
配置 ADC 定时器	ADCTimerConfig()	main()	adc.c
允许 ADC 定时器	ADCTimerEnable()	main()	adc.c
允许 ADC	**ADCEnable ()**	main()	adc.c
允许 ADC 通道	**ADCChannelEnable()**	main()	adc.c
禁止 ADC 通道	ADCChannelDisable()	main()	adc.c
获取 ADC FIFO 阈值	ADCFIFOLvlGet()	main()	adc.c
读取 ADC FIFO 数据	**ADCFIFORead()**	main()	adc.c

4.2.2 ADC 使用简介

ADC 的使用比较简单，主要包括引脚复用配置、初始化 ADC 和 ADC FIFO 操作等。

1．引脚复用配置

引脚复用配置主要是配置 ADC 引脚 PinTypeADC()，API 声明如下。

```
void PinTypeADC(unsigned long ulPin, unsigned long ulPinMode);
```

2．初始化 ADC

初始化 ADC 主要包括允许 ADC ADCEnable()和允许 ADC 通道 ADCChannelEnable()等，API 声明如下。

```
void ADCEnable(unsigned long ulBase);
void ADCChannelEnable(unsigned long ulBase, unsigned long ulChannel);
```

3．ADC FIFO 操作

ADC FIFO 操作包括获取 ADC FIFO 阈值 ADCFIFOLvlGet()和读取 ADC FIFO 数据 ADCFIFORead()，API 声明如下。

```
unsigned char ADCFIFOLvlGet(unsigned long ulBase, unsigned long ulChannel);
unsigned long ADCFIFORead(unsigned long ulBase, unsigned long ulChannel);
```

4.3 NVIC 应用

接口数据传送控制方式有查询、中断和 DMA 等，中断是重要的接口数据传送控制方式。中断控制分为全局和局部两级，全局中断由 NVIC 控制，局部中断由外设控制。

嵌套向量中断控制器 NVIC 支持多个内部异常和多达 240 个外部中断。从广义上讲，异常和中断都是暂停正在执行的程序转去执行异常或中断处理程序，然后返回原来的程序继续执行。从狭义上讲，异常由内部事件引起，而中断由外部硬件产生。

异常和中断的处理与子程序调用有相似之处，但也有下列本质区别。

- 什么时候调用子程序是确定的，而什么时候产生异常和中断是不确定的。
- 子程序的起始地址由调用程序给出，而异常和中断程序起始地址存放在地址表中。
- 子程序的执行一般是无条件的，而异常和中断处理程序的执行要先使能。

CC3200 NVIC 具有下列特性。

- 确定的、快速的中断处理，总是 12 个时钟周期。
- 每个中断的优先级可编程（0～7，0 表示最高优先级）。
 - ✧ 3～8 位优先级配置。
 - ✧ 动态重新分配中断优先级。
 - ✧ 中断优先级分组允许选择中断优先级分组和中断子优先级的数量。
- 低延迟中断和异常处理。
- 中断信号电平和边沿检测。

CC3200 异常和中断参见附录 F。

4.3.1　NVIC 示例程序分析

NVIC 示例程序在“example\interrupt”目录中，Interrupt 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- systick_if.c：SysTick 驱动程序文件。
- uart_if.c：UART 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Interrupt 项目程序流程如图 4.24 所示。



图 4.24　Interrupt 项目程序流程图

Interrupt 项目的运行结果如图 4.25 所示。

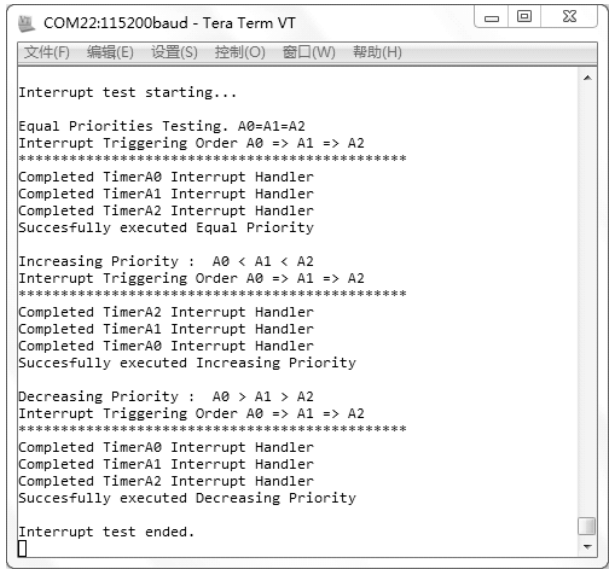


图 4.25　Interrupt 项目运行结果

下面重点介绍中断测试 DoInterruptTest()的分析步骤，主要包括下列内容。

- 中断测试初始化：InterruptTestInit()。
- 设置定时器。
- 优先级测试：PerformIntTest()。
- 中断测试结束：InterruptTestTerm()。
- 测试过程分析。

1．中断测试初始化 InterruptTestInit()

中断测试初始化 InterruptTestInit()包括允许和复位定时器、获取中断优先级和 SysTick 初始化 SysTickInit()等。

（1）允许和复位定时器。允许和复位定时器分别通过调用 PRCMPeripheralClkEnable()和 PRCMPeripheralReset()实现。有关定时器的函数调用在 GPT 应用中已经介绍，这里不再赘述。

（2）获取中断优先级。获取中断优先级分别通过调用 IntPriorityGet()和 IntPriorityGroupingGet()获取定时器中断优先级和中断优先级分组。

① 获取定时器中断优先级 IntPriorityGet()。获取定时器中断优先级 IntPriorityGet()的函数格式为

```
long IntPriorityGet(unsigned long ulInterrupt);
```

功能：获取中断优先级。

参数说明：ulInterrupt，中断号（2～193），在 hw_ints.h 中定义。

返回值：中断优先级（0x00～0xE0），在 interrupt.h 中定义。

IntPriorityGet()的声明在 interrupt.h 中，代码在 interrupt.c 中。TimerA0A 和 TimerA1A 的中断优先级 g_ulTimer0Apriority 和 g_ulTimer1APriority 均为 0。

② 获取中断优先级分组 IntPriorityGroupingGet()。获取中断优先级分组 IntPriorityGroupingGet()的函数格式为

```
unsigned long IntPriorityGroupingGet(void);
```

功能：获取中断优先级分组。

返回值：中断优先级分组。

IntPriorityGroupingGet()的声明在 interrupt.h 中，代码在 interrupt.c 中。中断优先级分组 g_lPriorityGrouping 的值为 7。

（3）SysTick 初始化 SysTickInit()。SysTick 初始化 SysTickInit()包括允许 SysTick 中断 SysTickIntEnable()、设置 SysTick 周期 SysTickPeriodSet()和允许 SysTick SysTickEnable()等，程序代码在 systick_if.c 中。

其中 SysTickPeriodSet()将 SysTick 的重装值设置为 0x400000（4194304），定时周期为 52 ms（4194304/80000000=52.4288 ms，误差值为 0.4288 ms×80000000=34304）。

SysTick 的中断处理程序为 SysTickHandler()，程序代码在 systick_if.c 中。

2．设置定时器

设置定时器通过调用 TimerConfigure()、TimerLoadSet()和 TimerIntRegister()分别将 TimerA0、TimerA1 和 TimerA2 设置为 TIMER_CFG_ONE_SHOT（32 位单次定时器），初值为 500 μs（40000/80000000），中断处理程序依次为 TimerA0IntHandler()、TimerA1IntHandler()和 TimerA2IntHandler()。

3．优先级测试 PerformIntTest()

优先级测试通过调用 PerformIntTest()依次完成相等优先级测试、增序优先级测试和减序优先级测试，PerformIntTest()的函数格式为。

```
tBoolean PerformIntTest(unsigned long ulPriBits, unsigned char ucPriorityA0,
                        unsigned char ucPriorityA1,
                        unsigned char ucPriorityA2);
```

功能：完成中断测试。

参数说明：

- ulPriBits：中断优先级分组，设置值是 3。
- ucPriorityA0：TimerA0 中断优先级。
- ucPriorityA1：TimerA1 中断优先级。
- ucPriorityA2：TimerA2 中断优先级。

返回值：true 表示成功，false 表示超时。

相等优先级测试、增序优先级测试和减序优先级测试时 TimerA0A、TimerA1A 和 TimerA2A 的中断优先级设置分别为

- 相等优先级测试：LOW_PRIORITY　LOW_PRIORITY　LOW_PRIORITY。
- 增序优先级测试：LOW_PRIORITY　MIDDLE_PRIORITY　HIGH_PRIORITY。
- 减序优先级测试：HIGH_PRIORITY　MIDDLE_PRIORITY　LOW_PRIORITY。

PerformIntTest()包括设置中断优先级、清除定时器中断、允许定时器中断、允许定时器、等待测试完成 UTUtilsWaitForCount()和禁止定时器等。

（1）设置中断优先级。设置中断优先级分别通过调用 IntPriorityGroupingSet()和 IntPrioritySet()设置中断优先级分组和定时器中断优先级。

① 设置中断优先级分组 IntPriorityGroupingSet()。设置中断优先级分组 IntPriorityGroupingSet()的函数格式为

```
void IntPriorityGroupingSet(unsigned long ulBits);
```

功能：设置中断优先级分组。

参数说明：ulBits，中断优先级分组，设置值是 3。

IntPriorityGroupingSet()的声明在 interrupt.h 中，代码在 interrupt.c 中。

② 设置定时器中断优先级 IntPrioritySet()。设置定时器中断优先级 IntPrioritySet()的函数格式为

```
void IntPrioritySet(unsigned long ulInterrupt, unsigned char ucPriority);
```

功能：设置中断优先级。

参数说明：

- ulInterrupt：中断号（2～193），在 hw_ints.h 中定义。
- ucPriority：中断优先级（0x00～0xE0），在 interrupt.h 中定义。

IntPrioritySet()的声明在 interrupt.h 中，代码在 interrupt.c 中。

（2）清除定时器中断。清除定时器中断通过调用 TimerIntStatus()和 TimerIntClear()获取定时器的中断状态并清除定时器中断。

（3）允许定时器中断。允许定时器中断通过调用 TimerIntEnable()实现。

（4）允许定时器。允许定时器通过调用 TimerEnable()允许 TIMERA0。

（5）等待测试完成 UTUtilsWaitForCount()。等待测试完成通过调用 UTUtilsWaitForCount()实现，函数格式为

```
tBoolean UTUtilsWaitForCount(volatile unsigned long *pulCount,
                             unsigned long ulTarget,
                             unsigned long ulTimeoutMs);
```

功能：等待计数到目标值。

参数说明：

- pulCount：计数指针，指向 TimerA0 中断计数 g_ulA0IntCount。
- ulTarget：目标值，传递值是 1。
- ulTimeoutMs：超时值。

返回值：true 表示计数到目标值，false 表示超时。

UTUtilsWaitForCount()的声明在 systick_if.h 中，代码在 systick_if.c 中。UTUtilsWaitForCount()通过调用 UTUtilsGetSysTime()获取系统（SysTick）时间，UTUtilsGetSysTime()又通过调用 SysTickValueGet()获取 SysTick 当前值。

（6）禁止定时器。禁止定时器通过调用 TimerDisable()和 TimerIntDisable()实现。

4．中断测试结束 InterruptTestTerm()

中断测试结束 InterruptTestTerm()包括注销定时器中断、恢复中断优先级、复位和禁止定时器等。

5．测试过程分析

下面分两种情况对测试过程进行分析。

（1）相等优先级或减序优先级测试。允许 TIMERA0 500 μs 后产生 TimerA0 中断，执行 TimerA0 中断处理程序 TimerA0Int Handler()：清除 TimerA0 中断，允许 TimerA1 并延时 2 ms。允许 TimerA1 在 500 μs 后产生 TimerA1 中断。

对于相等优先级或减序优先级测试，TimerA1 的中断优先级等于或小于 TimerA0 的中断优先级，TimerA1 中断不能中断正在执行的 TimerA0 中断处理程序，TimerA0 中断处理程序

继续执行：ulCountOnEntry 和 g_ulA1IntCount 相等，g_bA1CountChanged 赋值 false，g_ulA0IntCount 加 1，超级终端显示下列信息。

```
Completed TimerA0 Interrupt Handler          完成 TimerA0 中断处理
```

TimerA0 中断处理完成后执行 TimerA1 中断处理程序 TimerA1IntHandler()，TimerA1 中断处理的过程和 TIMERA0 中断处理过程相似：清除 TimerA1 中断，允许 TimerA2 并延时 2 ms。允许 TimerA2 在 500 μs 后产生 TimerA2 中断。

同样，对于相等优先级或减序优先级测试，TimerA2 的中断优先级等于或小于 TimerA1 的中断优先级，TimerA2 中断不能中断正在执行的 TimerA1 中断处理程序，TimerA1 中断处理程序继续执行：ulCountOnEntry 和 g_ulA2IntCount 相等，g_bA2CountChanged 赋值 false，g_ulA1IntCount 加 1，超级终端显示下列信息。

```
Completed TimerA1 Interrupt Handler          完成 TimerA1 中断处理
```

TimerA1 中断处理完成后执行 TimerA2 中断处理程序 TimerA2IntHandler()，TimerA2 中断处理过程比较简单：清除 TimerA2 中断，g_ulA2IntCount 加 1，超级终端显示下列信息。

```
Completed TimerA2 Interrupt Handler          完成 TimerA2 中断处理
```

相等优先级或减序优先级测试完成后，g_ulA0IntCount、g_ulA1IntCount 和 g_ulA2IntCount 均为 1，g_bA1CountChanged 和 g_bA2CountChanged 均为 false。

（2）增序优先级测试。对于增序优先级测试，TimerA1 的中断优先级大于 TimerA0 的中断优先级，TimerA1 中断将中断正在执行的 TimerA0 中断处理程序，转去执行 TimerA1 中断处理程序：清除 TimerA1 中断，允许 TimerA2 并延时 2 ms。允许 TimerA2 在 500 μs 后产生 TimerA2 中断。

同样，对于增序优先级测试，TimerA2 的中断优先级大于 TimerA1 的中断优先级，TimerA2 中断将中断正在执行的 TimerA1 中断处理程序，转去执行 TimerA2 中断处理程序：清除 TimerA2 中断，g_ulA2IntCount 加 1，超级终端显示下列信息。

```
Completed TimerA2 Interrupt Handler          完成 TimerA2 中断处理
```

TimerA2 中断处理完成后继续执行 TimerA1 中断处理：ulCountOnEntry 和 g_ulA2IntCount 不相等，g_bA2CountChanged 赋值 true，g_ulA1IntCount 加 1，超级终端显示下列信息。

```
Completed TimerA1 Interrupt Handler          完成 TimerA1 中断处理
```

同样，TIMERA1 中断处理完成后继续执行 TimerA0 中断处理：ulCountOnEntry 和 g_ulA1IntCount 不相等，g_bA1CountChanged 赋值 true，g_ulA0IntCount 加 1，超级终端显示下列信息。

```
Completed TimerA0 Interrupt Handler          完成 TimerA0 中断处理
```

增序优先级测试完成后，g_ulA0IntCount、g_ulA1IntCount 和 g_ulA2IntCount 均为 1，g_bA1CountChanged 和 g_bA2CountChanged 均为 true。

因为 TimerA1 的中断优先级比 TimerA0 的中断优先级高，所以在 TimerA0 中断处理过程中，TimerA1 中断正在执行的 TimerA0 中断处理程序，转去执行 TimerA1 中断处理程序。同样，TimerA2 中断正在执行的 TimerA1 中断处理程序，转去执行 TimerA2 中断处理程序。

TimerA2 中断处理完成后继续执行 TimerA1 中断处理程序，TimerA1 中断处理完成后继续执行 TimerA0 中断处理程序。因此，超级终端显示提示信息的顺序和相等优先级或减序优先级测试的顺序相反。

综上所述，Interrupt 项目使用的与中断有关的 API 函数如表 4.5 所示。

表 4.5　Interrupt 项目使用的与中断有关的 API 函数

功　能	函数名	调用函数	代码文件
获取中断优先级分组	IntPriorityGroupingGet()	InterruptTestInit()	interrupt.c
获取中断优先级	IntPriorityGet()	InterruptTestInit()	interrupt.c
设置中断优先级分组	IntPriorityGroupingSet()	PerformIntTest()、InterruptTestTerm()	interrupt.c
设置中断优先级	IntPrioritySet()	PerformIntTest()、InterruptTestTerm()	interrupt.c
注册定时器中断	**TimerIntRegister()**	DoInterruptTest()	timer.c
允许定时器中断	**TimerIntEnable()**	PerformIntTest()	timer.c
获取定时器中断状态	TimerIntStatus()	PerformIntTest()、TimerA0IntHandler()、TimerA1IntHandler()、TimerA2IntHandler()	timer.c
清除定时器中断	TimerIntClear()	PerformIntTest()、TimerA0IntHandler()、TimerA1IntHandler()、TimerA2IntHandler()	timer.c
禁止定时器中断	TimerIntDisable()	PerformIntTest()	timer.c

4.3.2　NVIC 使用简介

NVIC 的使用包括中断初始化、中断优先级操作和中断操作等。

1．中断初始化

中断初始化包括设置中断向量表基地址 IntVTableBaseSet()和允许处理器中断 IntMasterEnable()，API 声明如下。

```
void IntVTableBaseSet(unsigned long ulVtableBase);
tBoolean IntMasterEnable(void);
```

2．中断优先级操作

中断优先级操作包括获取中断优先级分组 IntPriorityGroupingGet()、获取中断优先级 IntPriorityGet()、设置中断优先级分组 IntPriorityGroupingSet()和获取中断优先级 IntPrioritySet()，API 声明如下。

```
unsigned long IntPriorityGroupingGet(void);
long IntPriorityGet(unsigned long ulInterrupt);
void IntPriorityGroupingSet(unsigned long ulBits);
void IntPrioritySet(unsigned long ulInterrupt, unsigned char ucPriority);
```

NVIC 支持 256 个（8 位）中断优先级，256 个中断优先级可编程分为若干组，每组包含若干子优先级。NVIC 中断优先级分组设置如表 4.6 所示。

表 4.6　NVIC 中断优先级分组设置

组　　号	优先级分组位段	子优先级位段	优先级分组数	子优先级数
0	无	[7:0]	1	256
1	[7]	[6:0]	2	128
2	[7:6]	[5:0]	4	64
3	[7:5]	[4:0]	8	32
4	[7:4]	[3:0]	16	16
5	[7:3]	[2:0]	32	8
6	[7:2]	[1:0]	64	4
7	[7:1]	[0]	128	2

CC3200 支持 8 个中断优先级，使用 8 位中断优先级的高 3 位，中断优先级分组设置如表 4.7 所示。

表 4.7　CC3200 中断优先级分组设置

组　　号	优先级分组位段	子优先级位段	优先级分组数	子优先级数
0	无	[7:5]	1	8
1	[7]	[6:5]	2	4
2	[7:6]	[5]	4	2
3	[7:5]	无	8	1

优先级分组高（数值小）的中断可以中断优先级分组低（数值大）的中断，而子优先级高的中断不能中断子优先级低的中断，因此，要想使新中断能够中断当前正在执行的中断处理，新中断的优先级分组必须比正在执行中断的优先级分组高（数值小）。

3．中断操作

中断操作包括注册中断 IntRegister()和允许中断 IntEnable()，API 声明如下。

```
void IntRegister(unsigned long ulInterrupt, void (*pfnHandler)(void));
void IntEnable(unsigned long ulInterrupt);
```

4.3.3　外设中断使用小结

外设中断使用包括外设中断初始化和外设中断操作，外设中断初始化包括注册外设中断和允许外设中断，外设中断操作包括获取外设中断状态和清除外设中断标志。

不同外设的中断操作大同小异，其中最主要的区别是中断标志不同。下面分别介绍 GPIO、UART、SPI、I2C、GPT 和 DAC 等外设中断的使用。

1．GPIO 中断使用

GPIO 中断使用的 API 声明如下。

```
void GPIOIntTypeSet(unsigned long ulPort, unsigned char ucPins,
                    unsigned long ulIntType);
void GPIOIntRegister(unsigned long ulPort, void (*pfnHandler)(void));
void GPIOIntEnable(unsigned long ulPort, unsigned long ulIntFlags);
```

```
unsigned long GPIOIntStatus(unsigned long ulPort, tBoolean bMasked);
void GPIOIntClear(unsigned long ulPort, unsigned long ulIntFlags);
```

参数说明：

- ulPort：GPIO 端口基地址，在 hw_memmap.h 中定义。
- pfnHandler：中断处理程序指针。
- ucPins：引脚位权，取值为 0x01、0x02、0x04、0x08、0x10、0x20、0x40 或 0x80。
- ulIntType：中断类型，包括下列值。
 - ✧ GPIO_FALLING_EDGE：下降沿中断，0x0000 0000。
 - ✧ GPIO_RISING_EDGE：上升沿中断，0x0000 0004。
 - ✧ GPIO_BOTH_EDGES：双沿中断，0x0000 0001。
 - ✧ GPIO_LOW_LEVEL：低电平中断，0x0000 0002。
 - ✧ GPIO_HIGH_LEVEL：高电平中断，0x0000 0006。
- ulIntFlags：中断标志，包括下列值。
 - ✧ GPIO_INT_DMA：DMA 完成中断，0x0000 0100。
 - ✧ GPIO_INT_PIN_7：PIN_7 中断，0x0000 0080。
 - ✧ GPIO_INT_PIN_6：PIN_6 中断，0x0000 0040。
 - ✧ GPIO_INT_PIN_5：PIN_5 中断，0x0000 0020。
 - ✧ GPIO_INT_PIN_4：PIN_4 中断，0x0000 0010。
 - ✧ GPIO_INT_PIN_3：PIN_3 中断，0x0000 0008。
 - ✧ GPIO_INT_PIN_2：PIN_2 中断，0x0000 0004。
 - ✧ GPIO_INT_PIN_1：PIN_1 中断，0x0000 0002。
 - ✧ GPIO_INT_PIN_0：PIN_0 中断，0x0000 0001。

2．UART 中断使用

UART 中断使用的 API 声明如下。

```
void UARTIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void UARTIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long UARTIntStatus(unsigned long ulBase, tBoolean bMasked);
void UARTIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

参数说明：

- ulBase：UART 基地址，在 hw_memmap.h 中定义。
- pfnHandler：中断处理程序指针。
- ulIntFlags：中断标志，包括下列值。
 - ✧ UART_INT_DMATX：DMA 发送中断，0x0002 0000。
 - ✧ UART_INT_DMARX：DMA 接收中断，0x0001 0000。
 - ✧ UART_INT_EOT：传输结束中断，0x0000 0800。
 - ✧ UART_INT_OE：超限错误中断，0x0000 0400。
 - ✧ UART_INT_BE：间断错误中断，0x0000 0200。
 - ✧ UART_INT_PE：校验错误中断，0x0000 0100。

- ✧ UART_INT_FE：帧错误中断，0x0000 0080。
- ✧ UART_INT_RT：接收超时中断，0x0000 0040。
- ✧ UART_INT_TX：发送中断，0x0000 0020。
- ✧ UART_INT_RX：接收中断，0x0000 0010。
- ✧ UART_INT_CTS：清除发送中断，0x0000 0002。

3．SPI 中断使用

SPI 中断使用的 API 声明如下。

```
void SPIIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void SPIIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long SPIIntStatus(unsigned long ulBase, tBoolean bMasked);
void SPIIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

参数说明：

- ulBase：SPI 基地址，GSPI 的基地址是 0x44021000，在 hw_memmap.h 中定义。
- pfnHandler：中断处理程序指针。
- ulIntFlags：中断标志，包括下列值。
 - ✧ SPI_INT_DMATX：DMA 发送中断，0x2000 0000。
 - ✧ SPI_INT_DMARX：DMA 接收中断，0x1000 0000。
 - ✧ SPI_INT_EOW：字计数结束中断，0x0002 0000。
 - ✧ SPI_INT_WKS：唤醒中断，0x0001 0000。
 - ✧ SPI_INT_RX_OVRFLOW：接收溢出中断，0x0000 0008。
 - ✧ SPI_INT_RX_FULL：接收缓存区满中断，0x0000 0004。
 - ✧ SPI_INT_TX_UDRFLOW：发送溢出中断，0x0000 0002。
 - ✧ SPI_INT_TX_ EMPTY：发送缓存区空中断，0x0000 0001。

4．I2C 中断使用

I2C 中断使用的 API 声明如下。

```
void I2CIntRegister(uint32_t ui32Base, void(*pfnHandler)(void));
void I2CMasterIntEnableEx(uint32_t ui32Base, uint32_t ui32IntFlags);
long I2CMasterIntStatusEx(uint32_t ui32Base, bool bMasked);
void I2CMasterIntClearEx(uint32_t ui32Base, uint32_t ui32IntFlags);
```

参数说明：

- ui32Base：I2C 基地址，I2C 的基地址是 0x4002 0000，在 hw_memmap.h 中定义。
- pfnHandler：中断处理程序指针。
- ui32IntFlags：中断标志，包括下列值。
 - ✧ I2C_MASTER_INT_RX_FIFO_FULL：接收 FIFO 满中断，0x0000 0800。
 - ✧ I2C_MASTER_INT_TX_FIFO_EMPTY：发送 FIFO 空中断，0x0000 0400。
 - ✧ I2C_MASTER_INT_RX_FIFO_REQ：接收 FIFO 请求中断，0x0000 0200。
 - ✧ I2C_MASTER_INT_TX_FIFO_REQ：发送 FIFO 请求中断，0x0000 0100。

- ✧ I2C_MASTER_INT_ARB_LOST：仲裁丢失中断，0x0000 0080。
- ✧ I2C_MASTER_INT_STOP：停止中断，0x0000 0040。
- ✧ I2C_MASTER_INT_START：起始中断，0x0000 0020。
- ✧ I2C_MASTER_INT_NACK：应答中断，0x0000 0010。
- ✧ I2C_MASTER_INT_TX_DMA_DONE：发送 DMA 完成中断，0x0000 0008。
- ✧ I2C_MASTER_INT_RX_DMA_DONE：接收 DMA 完成中断，0x0000 0004。
- ✧ I2C_MASTER_INT_TIMEOUT：超时中断，0x0000 0002。
- ✧ I2C_MASTER_INT_DATA：数据中断，0x0000 0001。

5．GPT 中断使用

GPT 中断使用的 API 声明如下。

```
void TimerIntRegister(unsigned long ulBase, unsigned long ulTimer,
                      void (*pfnHandler)(void));
void TimerIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long TimerIntStatus(unsigned long ulBase, tBoolean bMasked);
void TimerIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

参数说明：

- ulBase：定时器基地址，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，包括下列值。
 - ✧ TIMER_A：定时器 A，0x0000 00FF。
 - ✧ TIMER_B：定时器 B，0x0000 FF00。
 - ✧ TIMER_BOTH：定时器 A 和定时器 B，0x0000 FFFF。
- pfnHandler：中断处理程序指针。
- ulIntFlags：中断标志，包括下列值。
 - ✧ TIMER_TIMB_DMA：定时器 B DMA 中断，0x0000 2000。
 - ✧ TIMER_TIMB_MATCH：定时器 B 匹配中断，0x0000 0800。
 - ✧ TIMER_CAPB_EVENT：定时器 B 捕捉事件中断，0x0000 0400。
 - ✧ TIMER_CAPB_MATCH：定时器 B 捕捉匹配中断，0x0000 0200。
 - ✧ TIMER_TIMB_TIMEOUT：定时器 B 超时中断，0x0000 0100。
 - ✧ TIMER_TIMA_DMA：定时器 A DMA 中断，0x0000 0020。
 - ✧ TIMER_TIMA_MATCH：定时器 A 匹配中断，0x0000 0010。
 - ✧ TIMER_CAPA_EVENT：定时器 A 捕捉事件中断，0x0000 0004。
 - ✧ TIMER_CAPA_MATCH：定时器 A 捕捉匹配中断，0x0000 0002。
 - ✧ TIMER_TIMA_TIMEOUT：定时器 A 超时中断，0x0000 0001。

6．ADC 中断使用

ADC 中断使用的 API 声明如下。

```
void ADCIntRegister(unsigned long ulBase, unsigned long ulChannel,
                    void (*pfnHandler)(void));
void ADCIntEnable(unsigned long ulBase, unsigned long ulChannel,
```

```
                    unsigned long ulIntFlags);
unsigned long ADCIntStatus(unsigned long ulBase, unsigned long ulChannel);
void ADCIntClear(unsigned long ulBase, unsigned long ulChannel,
                 unsigned long ulIntFlags);
```

参数说明：

- ulBase：ADC 基地址，ADC 的基地址是 0x4402 E800，在 hw_memmap.h 中定义。
- ulChannel：ADC 通道，取值为 ADC_CH_0～3（ADC 通道 0～3）。
- pfnHandler：中断处理程序指针。
- ulIntFlags：中断标志，包括下列值。
 - ADC_DMA_DONE：DMA 完成，0x0000 0010。
 - ADC_FIFO_OVERFLOW：FIFO 上溢出，0x0000 0008。
 - ADC_FIFO_UNDERFLOW：FIFO 下溢出，0x0000 0004。
 - ADC_FIFO_EMPTY：FIFO 空，0x0000 0002。
 - ADC_FIFO_FULL：FIFO 满，0x0000 0001。

4.4　μDMA 应用

直接存储器存取（DMA）可以在外设和存储器之间或者存储器和存储器之间进行批量数据的传输，其传送过程中无须 CPU 干预，数据可以通过 DMA 快速地传送，这就可以节省 CPU 的资源来进行其他操作。

CC3200 包含的 DMA 称为μDMA（微型 DMA），μDMA 具有下列特性。

- 32 个可配置通道。
- 支持存储器到存储器、存储器到外设，以及外设到存储器传输模式。
 - 基本的简单传输。
 - 乒乓连续数据流传输。
 - 分集单次请求多达 256 个可编程列表传输。
- 高度灵活和可配置的通道操作。
 - 独立的配置和操作通道。
 - 支持片内模块的专用通道。
 - 通道的发送和接收路径支持双向模块。
 - 支持软件启动传输的专用通道。
 - 可选的软件启动请求。
- 两级优先级。
- 提高μDMA 和处理器内核间总线访问性能的优化设计。
- 8 位、16 位或 32 位数据宽度。
- 传输长度 1～1024 可编程。
- 源和目的地址可按字节、半字和字增加或不增加。
- 通道单独的中断可在传输完成时产生中断。

μDMA 通道分配参见附录 G。

4.4.1 UART DMA 示例程序分析

UART DMA 示例程序在“example\uart_dma”目录中，Uart_dma 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- udma_if.c：μDMA 驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Uart_dma 项目程序流程如图 4.26 所示。

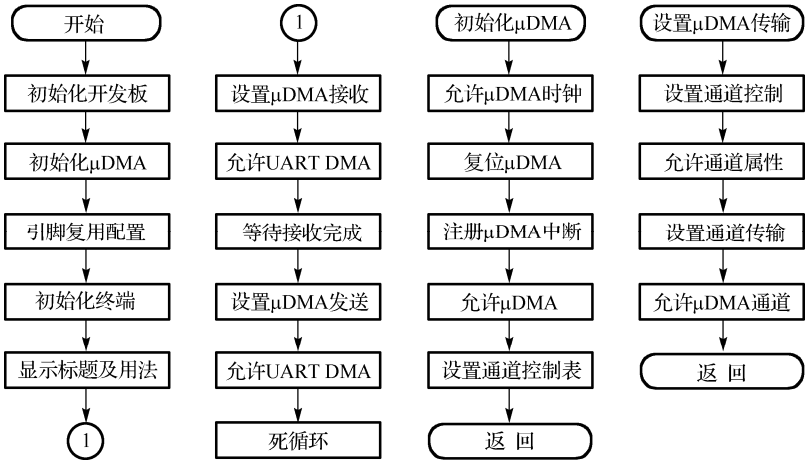


图 4.26 Uart_dma 项目程序流程图

Uart_dma 项目的运行结果如图 4.27 所示。

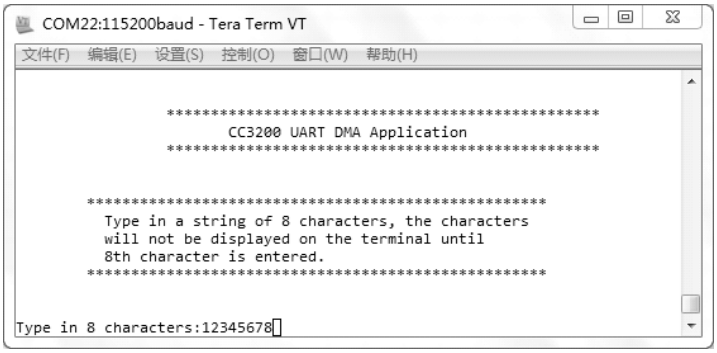


图 4.27 Uart_dma 项目运行结果

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化μDMA：UDMAInit()。
- 设置μDMA 传输：UDMASetupTransfer()。
- UART DMA 操作。

1．初始化μDMA 函数 UDMAInit()

初始化μDMA 函数 UDMAInit()包括允许μDMA 时钟、复位μDMA、注册μDMA 中断、允许μDMA 和设置μDMA 通道控制表基地址等，相关程序代码在 udma_if.c 中。

（1）允许μDMA 时钟：通过调用 PRCMPeripheralClkEnable()实现。

（2）复位μDMA：通过调用 PRCMPeripheralReset()实现。

（3）注册μDMA 中断：通过调用 uDMAIntRegister()实现。只有软件μDMA 传输（通道编码为 3）时才使用μDMA 中断，外设μDMA 传输使用对应的外设中断。uDMAIntRegister()又调用 IntRegister()和 IntEnable()实现μDMA 软件中断和μDMA 错误中断的注册与允许，μDMA 软件中断和μDMA 错误中断的处理程序分别是 DmaSwInt Handler()和 DmaErrorIntHandler()。

（4）允许μDMA：通过调用 uDMAEnable()实现。

（5）设置μDMA 通道控制表基地址：通过调用 uDMAControlBaseSet()实现，函数格式为

```
void uDMAControlBaseSet(void *pControlTable);
```

功能：设置μDMA 通道控制表基地址。

参数说明：

*pControlTable：μDMA 通道控制表基地址指针，μDMA 通道控制表包含μDMA 通道的下列内容。

- pvSrcEndAddr：源结束地址指针。
- pvDstEndAddr：目的结束地址指针。
- ulControl：通道控制参数。
- ulSpare：备用。

通道控制参数包括传输模式、传输项大小、仲裁大小、源地址增量和目的地址增量等。

① 通道模式包含下列值。

- UDMA_MODE_STOP：停止模式，0x0000 0000。
- UDMA_MODE_BASIC：基本模式，0x0000 0001。
- UDMA_MODE_AUTO：自动模式，0x0000 0002。
- UDMA_MODE_PINGPONG：乒乓模式，0x0000 0003。
- UDMA_MODE_MEM_SCATTER_GATHER：存储器分集模式，0x0000 0004。
- UDMA_MODE_PER_SCATTER_GATHER：外设分集模式，0x0000 0006。

基本模式用于外设和存储器之间的 DMA 传输；自动模式和基本模式相似，用于存储器和存储器之间的软件 DMA 传输；乒乓模式用于外设和存储器之间的连续 DMA 传输，乒乓模式需要主和备用两个通道控制，系统使用主和备用两个通道控制交替进行 DMA 传输；分集模式包括存储器分集模式和外设分集模式，分集模式将两个及两个以上数据块利用 DMA 传输到一个数据块或外设。

② 传输项大小包含下列值。

- UDMA_SIZE_8：8 位，0x0000 0000。

- UDMA_SIZE_16：16 位，0x1100 0000。
- UDMA_SIZE_32：32 位，0x2200 0000。

③ 仲裁大小包含下列值。

- UDMA_ARB_1：1 项，0x0000 0000。
- UDMA_ARB_2：2 项，0x0000 4000。
- UDMA_ARB_4：4 项，0x0000 8000。
- UDMA_ARB_8：8 项，0x0000 C000。
- UDMA_ARB_16：16 项，0x0001 0000。
- UDMA_ARB_32：32 项，0x0001 4000。
- UDMA_ARB_64：64 项，0x0001 8000。
- UDMA_ARB_128：128 项，0x0001 C000。
- UDMA_ARB_256：256 项，0x0002 0000。
- UDMA_ARB_512：512 项，0x0002 4000。
- UDMA_ARB_1024：1024 项，0x0002 8000。

④ 源地址增量包含下列值。

- UDMA_SRC_INC_8：8 位，0x0000 0000。
- UDMA_SRC_INC_16：16 位，0x0400 0000。
- UDMA_SRC_INC_32：32 位，0x0800 0000。
- UDMA_SRC_INC_NONE：不增加，0x0C00 0000。

⑤ 目的地址增量包含下列值。

- UDMA_DST_INC_8：8 位，0x0000 0000。
- UDMA_DST_INC_16：16 位，0x4000 0000。
- UDMA_DST_INC_32：32 位，0x8000 0000。
- UDMA_DST_INC_NONE：不增加，0xC000 0000。

uDMAControlBaseSet()的声明在 udma.h 中，代码在 udma.c 中。

2．设置μDMA 传输 UDMASetupTransfer()

UART 接收和发送的设置μDMA 传输调用语句分别为

```
UDMASetupTransfer(UDMA_CH8_UARTA0_RX, UDMA_MODE_BASIC, 8, UDMA_SIZE_8,
                  UDMA_ARB_2, (void *)(UARTA0_BASE+UART_O_DR),
                  UDMA_SRC_INC_NONE,(void *)ucTextBuff, UDMA_DST_INC_8);

UDMASetupTransfer(UDMA_CH9_UARTA0_TX, UDMA_MODE_BASIC, 8, UDMA_SIZE_8,
                  UDMA_ARB_2, (void *)ucTextBuff, UDMA_SRC_INC_8,
                  (void *)(UARTA0_BASE+UART_O_DR), UDMA_DST_INC_NONE);
```

设置μDMA 传输 UDMASetupTransfer()的函数内容如图 4.28 所示。

```
void UDMASetupTransfer(unsigned long ulChannel, unsigned long ulMode,
                       unsigned long ulItemCount, unsigned long ulItemSize,
```

```
                                unsigned long ulArbSize, void *pvSrcBuf,
                                unsigned long ulSrcInc, void *pvDstBuf,
                                unsigned long ulDstInc);
```

```
353 void UDMASetupTransfer(unsigned long ulChannel, unsigned long ulMode,
354                          unsigned long ulItemCount, unsigned long ulItemSize,
355                          unsigned long ulArbSize, void *pvSrcBuf,
356                          unsigned long ulSrcInc, void *pvDstBuf,
357                          unsigned long ulDstInc)
358 {
359     MAP_uDMAChannelControlSet(ulChannel, ulItemSize | ulSrcInc | ulDstInc | ulArbSize);
360     MAP_uDMAChannelAttributeEnable(ulChannel,UDMA_ATTR_USEBURST);
361     MAP_uDMAChannelTransferSet(ulChannel, ulMode, pvSrcBuf, pvDstBuf, ulItemCount);
362     MAP_uDMAChannelEnable(ulChannel);
363 }
```

图 4.28　Uart_dma 项目 UDMASetupTransfer()函数内容

功能：设置μDMA 传输。

参数说明：

- ulChannel：通道选择，传递值分别是 UDMA_CH8_UARTA0_RX 和 UDMA_CH9_UARTA0_TX，在 udma.h 中分别定义为 0x0000 0008 和 0x0000 0009，包含通道号（0～31）、通道编码（0～3）和下列值。
 - ✧ UDMA_PRI_SELECT：主选择，0x0000 0000。
 - ✧ UDMA_ALT_SELECT：备用选择，0x0000 0020。
- ulMode：通道模式，传递值是 UDMA_MODE_BASIC（基本模式）。
- ulItemCount：传输项数，传递值是 8。
- ulItemSize：传输项大小，传递值是 UDMA_SIZE_8（8 位）。
- ulArbSize：仲裁大小，传递值是 UDMA_ARB_2（2 项）。
- *pvSrcBuf：源缓存地址指针，接收时指向 UARTA0_BASE+UART_O_DR（UARTA0 数据寄存器），发送时指向 ucTextBuff。
- ulSrcInc：源地址增量，传递值分别是 UDMA_SRC_INC_NONE（无增量）和 UDMA_SRC_INC_8（8 位）。
- *pvDstBuf：目的缓存地址指针，接收时指向 ucTextBuff，发送时指向 UARTA0_BASE+UART_O_DR（UARTA0 数据寄存器）。
- ulDstInc：目的地址增量，传递值分别是 UDMA_DST_INC_8（8 位）和 UDMA_DST_INC_NONE（无增量）。

UDMASetupTransfer()的声明在 udma_if.h 中，代码在 udma_if.c 中。设置μDMA 传输 UDMASetupTransfer()主要包括下列内容。

- 设置μDMA 通道控制：uDMAChannelControlSet()。
- 允许μDMA 通道属性：uDMAChannelAttributeEnable()。
- 设置μDMA 通道传输：uDMAChannelTransferSet()。
- 允许μDMA 通道：uDMAChannelEnable()。

（1）设置μDMA 通道控制 uDMAChannelControlSet()。

```
void uDMAChannelControlSet(unsigned long ulChannelStructIndex,
                           unsigned long ulControl);
```

功能：设置μDMA 通道控制参数。

参数说明：

- ulChannelStructIndex：通道选择，传递值分别是 0x0000 0008，即通道 8 编码 0（UARTA0_ RX），以及 0x0000 0009，即通道 9 编码 0（UARTA0_TX）。
- ulControl：通道控制参数，包括传输项大小、仲裁大小、源地址增量和目的地址增量。

uDMAChannelControlSet()的声明在 udma.h 中，代码在 udma.c 中。

（2）允许μDMA 通道属性 uDMAChannelAttributeEnable()。

```
void uDMAChannelAttributeEnable(unsigned long ulChannelNum,
                                unsigned long ulAttr);
```

功能：允许μDMA 通道属性。

参数说明：

- ulChannelNum：通道选择，传递值分别是 0x0000 0008，即通道 8 编码 0（UARTA0_RX），以及 0x0000 0009，即通道 9 编码 0（UARTA0_TX）。
- ulAttr：通道属性，传递值是 UDMA_ATTR_USEBURST（突发模式）。ulAttr 包含下列值。
 - ✧ UDMA_ATTR_USEBURST：突发模式，0x0000 0001。
 - ✧ UDMA_ATTR_ALTSELECT：备选模式，0x0000 0002。
 - ✧ UDMA_ATTR_HIGH_PRIORITY：高优先级，0x0000 0004。
 - ✧ UDMA_ATTR_REQMASK：请求屏蔽，0x0000 0008。
 - ✧ UDMA_ATTR_ALL：所有属性，0x0000 000F。

uDMAChannelAttributeEnable()的声明在 udma.h 中，代码在 udma.c 中。

（3）设置μDMA 通道传输 uDMAChannelTransferSet()。

```
void uDMAChannelTransferSet(unsigned long ulChannelStructIndex,
                            unsigned long ulMode, void *pvSrcAddr,
                            void *pvDstAddr,unsigned long ulTransferSize);
```

功能：设置μDMA 通道传输参数。

参数说明：

- ulChannelStructIndex：通道选择，传递值分别是 0x0000 0008，即通道 8 编码 0（UARTA0_ RX），以及 0x0000 0009，即通道 9 编码 0（UARTA0_TX）。
- ulMode：通道模式，传递值是 UDMA_MODE_BASIC（基本模式）。
- *pvSrcAddr：源地址指针，接收时指向 UARTA0_BASE+UART_O_DR（UARTA0 数据寄存器），发送时指向 ucTextBuff。
- *pvDstAddr：目的地址指针，接收时指向 ucTextBuff，发送时指向 UARTA0_BASE + UART_O_DR（UARTA0 数据寄存器）。
- ulTransferSize：传输项数，传递值是 8。

uDMAChannelTransferSet()的声明在 udma.h 中，代码在 udma.c 中。

（4）允许μDMA 通道 uDMAChannelEnable()。

```
void uDMAChannelEnable(unsigned long ulChannelNum);
```

功能：允许μDMA 通道。

参数说明：ulChannelNum，通道选择，传递值分别是 0x0000 0008，即通道 8 编码 0（UARTA0_RX），以及 0x0000 0009，即通道 9 编码 0（UARTA0_TX）。

uDMAChannelEnable()的声明在 udma.h 中，代码在 udma.c 中。

3．UART DMA 操作

UART DMA 操作主要包括下列内容。

- 注册 UART 中断：UARTIntRegister()。
- 允许 UART 中断：UARTIntEnable()。
- 设置 UART FIFO 阈值：UARTFIFOLevelSet()。
- 允许 UART DMA 请求：UARTDMAEnable()。
- 禁止 UART DMA 请求：UARTDMADisable()。
- 清除 UART 中断：UARTIntClear()。

主程序首先调用 UARTIntRegister()注册 UART 中断，调用 UARTIntEnable()允许 UART DMA 接收中断（UART_INT_DMARX），调用 UARTFIFOLevelSet()将 UART 发送和接收 FIFO 的阈值都设为 1（16 位，两个字符）。

然后调用 UDMASetupTransfer()设置μDMA 接收，调用 UARTDMAEnable()允许 UART DMA 接收请求（UART_DMA_RX），等待接收完成（bRxDone = true）。调用 UDMASetupTransfer()设置μDMA 接收时的传输项数为 8，因此通过 UART 接收 8 个字符后产生 UART 中断，UART 中断处理程序 UARTIntHandler()调用 UARTDMADisable()禁止 UART DMA 接收请求，设置接收完成标志（bRxDone = true），调用 UARTIntClear()清除 UART 中断。

最后调用 UDMASetupTransfer()设置μDMA 发送，调用 UARTDMAEnable()允许 UART DMA 发送请求（UART_DMA_TX），通过 UART 发送 8 个字符后程序进入死循环。

注意：由于没有允许 UART DMA 发送中断，8 个字符发送完成后不会产生 UART 中断。另外，UART 中断处理程序 UARTIntHandler()没有对中断源进行判断，因此一旦由于其他原因产生中断进入中断处理程序，将在接收没有完成时禁止 UART DMA 接收请求，设置接收完成标志（bRxDone = true），结果异常。

综上所述，uart_dma 项目使用的与μDMA 有关的 API 函数如表 4.8 所示。

表 4.8　uart_dma 项目使用的与μDMA 有关的 API 函数

功　能	函数名	调用函数	代码文件
允许μDMA 时钟	**PRCMPeripheralClkEnable()**	UDMAInit()	prcm.c
复位μDMA	PRCMPeripheralReset()	UDMAInit()	prcm.c
注册μDMA 中断	uDMAIntRegister()	UDMAInit()	udma.c
允许μDMA	**uDMAEnable()**	UDMAInit()	udma.c
设置μDMA 通道控制表	uDMAControlBaseSet()	UDMAInit()	udma.c

续表

功　能	函数名	调用函数	代码文件
设置μDMA 传输	**UDMASetupTransfer()**	main()	udma_if.c
设置μDMA 通道控制	uDMAChannelControlSet()	UDMASetupTransfer()	udma.c
允许μDMA 通道属性	uDMAChannelAttributeEnable()	UDMASetupTransfer()	udma.c
设置μDMA 通道传输	uDMAChannelTransferSet()	UDMASetupTransfer()	udma.c
允许μDMA 通道	uDMAChannelEnable()	UDMASetupTransfer()	udma.c
注册 UART 中断	UARTIntRegister()	main()	uart.c
允许 UART 中断	UARTIntEnable()	main()	uart.c
设置 UART FIFO 阈值	UARTFIFOLevelSet()	main()	uart.c
允许 UART DMA 请求	UARTDMAEnable()	main()	uart.c
禁止 UART DMA 请求	UARTDMADisable()	UARTIntHandler()	uart.c
清除 UART 中断	UARTIntClear()	UARTIntHandler()	uart.c

4.4.2　μDMA 使用简介

μDMA 的使用包括μDMA 操作和外设 DMA 操作两部分。

1．μDMA 操作

μDMA 操作包括初始化μDMA UDMAInit()和设置μDMA 传输 UDMASetupTransfer()。

（1）初始化μDMA 函数 UDMAInit()。初始化μDMA 函数 UDMAInit()包括允许μDMA 时钟 PRCMPeripheralClkEnable()、复位μDMA 函数 PRCMPeripheralReset()、注册μDMA 中断 uDMAIntRegister()、允许μDMA 函数 uDMAEnable()和设置μDMA 通道控制表基地址 uDMAControlBaseSet()等，API 声明如下。

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
void PRCMPeripheralReset(unsigned long ulPeripheral);
void uDMAIntRegister(unsigned long ulIntChannel, void (*pfnHandler)(void));
void uDMAEnable(void);
void uDMAControlBaseSet(void *pControlTable);
```

（2）设置μDMA 传输 UDMASetupTransfer()。设置μDMA 传输 UDMASetupTransfer()包括设置μDMA 通道控制 uDMAChannelControlSet()、允许μDMA 通道属性 uDMAChannelAttributeEnable()、设置μDMA 通道传输 uDMAChannelTransferSet()和允许μDMA 通道 uDMAChannelEnable()等，API 声明如下。

```
void UDMASetupTransfer(unsigned long ulChannel, unsigned long ulMode,
                       unsigned long ulItemCount, unsigned long ulItemSize,
                       unsigned long ulArbSize, void *pvSrcBuf,
                       unsigned long ulSrcInc, void *pvDstBuf,
                       unsigned long ulDstInc);
void uDMAChannelControlSet(unsigned long ulChannelStructIndex,
                           unsigned long ulControl);
void uDMAChannelAttributeEnable(unsigned long ulChannelNum,
                                unsigned long ulAttr);
```

```
void uDMAChannelTransferSet(unsigned long ulChannelStructIndex,
                            unsigned long ulMode, void *pvSrcAddr,
                            void *pvDstAddr, unsigned long ulTransferSize);
void uDMAChannelEnable(unsigned long ulChannelNum);
```

μDMA 传输支持下列 6 种模式。

- UDMA_MODE_STOP：停止模式，0x0000 0000。
- UDMA_MODE_BASIC：基本模式，0x0000 0001。
- UDMA_MODE_AUTO：自动模式，0x0000 0002。
- UDMA_MODE_PINGPONG：乒乓模式，0x0000 0003。
- UDMA_MODE_MEM_SCATTER_GATHER：存储器分集模式，0x0000 0004。
- UDMA_MODE_PER_SCATTER_GATHER：外设分集模式，0x0000 0006。

Uart_dma 项目只使用了μDMA 传输的基本模式（UDMA_MODE_BASIC），其他模式的使用方法参见 4.4.3 节。

2．外设 DMA 操作

外设和存储器之间的 DMA 传输除了需要μDMA 操作外，还需要进行外设 DMA 操作。外设 DMA 操作包括外设 DMA 初始化和外设中断操作，外设 DMA 初始化包括允许外设 DMA、注册外设中断和允许外设 DMA 中断，外设中断操作包括获取外设中断状态和清除外设中断标志。

不同外设的 DMA 操作大同小异，其中最主要的区别是允许外设 DMA 和外设中断标志不同，下面分别介绍 GPIO、UART、SPI、GPT 和 DAC 等外设的 DMA 操作。

（1）GPIO DMA 操作。GPIO DMA 操作使用的 API 声明如下。

```
void GPIODMATriggerEnable(unsigned long ulPort);
void GPIOIntRegister(unsigned long ulPort, void (*pfnHandler)(void));
void GPIOIntEnable(unsigned long ulPort, unsigned long ulIntFlags);
```

与 DMA 有关的中断标志 ulIntFlags 是 GPIO_INT_DMA 表示 DMA 完成中断，0x0000 0100。

（2）UART DMA 操作。UART DMA 操作使用的 API 声明如下

```
void UARTDMAEnable(unsigned long ulBase, unsigned long ulDMAFlags);
void UARTIntRegister(unsigned long ulBase, void (*pfnHandler)(void));
void UARTIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
```

ulDMAFlags：DMA 标志，包含下列值。

- UART_DMA_RX：DMA 接收，0x0000 0001。
- UART_DMA_TX：DMA 发送，0x0000 0002。
- UART_DMA_ERR_RXSTOP：错误时停止接收，0x0000 0004。

与 DMA 有关的中断标志 ulIntFlags 如下。

- UART_INT_DMATX：DMA 发送中断，0x0002 0000。
- UART_INT_DMARX：DMA 接收中断，0x0001 0000。

（3）SPI DMA 操作。SPI DMA 操作使用的 API 声明如下。

```
void SPIDmaEnable(unsigned long ulBase, unsigned long ulFlags);
void SPIIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void SPIIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
```

ulFlags：DMA 标志，包含下列值。

- SPI_RX_DMA：DMA 接收，0x0000 8000。
- SPI_TX_DMA：DMA 发送，0x0000 4000。

与 DMA 有关的中断标志 ulIntFlags 如下。

- SPI_INT_DMATX：DMA 发送中断，0x2000 0000。
- SPI_INT_DMARX：DMA 接收中断，0x1000 0000。

（4）GPT DMA 操作。GPT DMA 操作使用的 API 声明如下。

```
void TimerDMAEventSet(unsigned long ulBase, unsigned long ulDMAEvent);
void TimerIntRegister(unsigned long ulBase, unsigned long ulTimer,
                      void (*pfnHandler)(void));
void TimerIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
```

ulDMAEvent：DMA 事件，包含下列值。

- TIMER_DMA_MODEMATCH_B：定时器 B 模式匹配，0x0000 0800。
- TIMER_DMA_CAPEVENT_B：定时器 B 捕捉事件，0x0000 0400。
- TIMER_DMA_CAPMATCH_B：定时器 B 捕捉匹配，0x0000 0200。
- TIMER_DMA_TIMEOUT_B：定时器 B 超时，0x0000 0100。
- TIMER_DMA_MODEMATCH_A：定时器 A 模式匹配，0x0000 0010。
- TIMER_DMA_CAPEVENT_A：定时器 A 捕捉事件，0x0000 0004。
- TIMER_DMA_CAPMATCH_A：定时器 A 捕捉匹配，0x0000 0002。
- TIMER_DMA_TIMEOUT_A：定时器 A 超时，0x0000 0001。

与 DMA 有关的中断标志 ulIntFlags 如下。

- TIMER_TIMB_DMA：定时器 B DMA 中断，0x0000 2000。
- TIMER_TIMA_DMA：定时器 A DMA 中断，0x0000 0020。

（5）ADC DMA 操作。ADC DMA 操作使用的 API 声明如下。

```
void ADCDMAEnable(unsigned long ulBase, unsigned long ulChannel);
void ADCIntRegister(unsigned long ulBase, unsigned long ulChannel,
                    void (*pfnHandler)(void));
void ADCIntEnable(unsigned long ulBase, unsigned long ulChannel,
                  unsigned long ulIntFlags);
```

与 DMA 有关的中断标志 ulIntFlags 是 ADC_DMA_DONE，表示 DMA 完成，0x0000 0010。

4.4.3 μDMA 示例程序分析

μDMA 示例程序在“example\udma”目录中，udma 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- udma_if.c：μDMA 驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

uDMA 项目程序流程如图 4.29 所示。

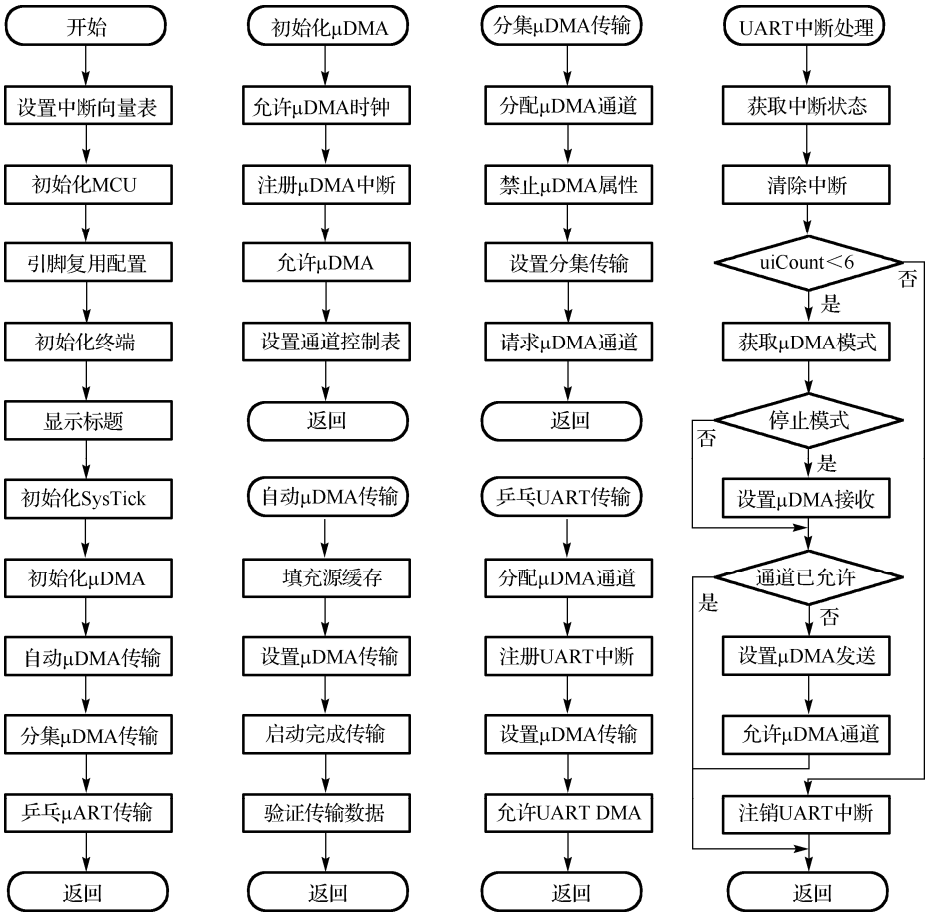


图 4.29　uDMA 项目程序流程图

uDMA 项目的运行结果如下。

```
Completed DMA Initialization                          完成 DMA 初始化

Starting Auto DMA Transfer                            开始自动 DMA 传输
Transfer complete                                     传输完成
Data transfered in Auto mode is verified              自动模式传输数据已验证

Starting Scatter Gather DMA Operation                 开始分集 DMA 操作
SG Mode Memory to Memory Transfer Completed           分集模式存储器到存储器传输完成
```

```
Completed Ping Pong Transfer from Memory to UART Peripheral
                                                    完成存储器到 UART 乒乓传输
Transfer Rate is 11520 Bytes/Sec                    传输速率是 11 520 B/s

Test Ended                                          测试结束
```

下面简要介绍程序的分析步骤，主要包括下列内容。

- 初始化μDMA：UDMAInit()。
- 自动μDMA 传输：InitSWTransfer()。
- 分集μDMA 传输：InitSGTransfer()。
- 乒乓 UART 传输。

1．初始化μDMA UDMAInit()

初始化μDMA UDMAInit()包括允许μDMA 时钟、复位μDMA、注册μDMA 中断、允许μDMA 和设置μDMA 通道控制表基地址等，相关程序代码在 udma_if.c 中。

初始化μDMA UDMAInit()的具体内容和 Uart_dma 项目相同。

2．自动μDMA 传输 InitSWTransfer()

自动μDMA 传输 InitSWTransfer()初始化μDMA 软件通道完成存储器到存储器μDMA 传输，包括设置μDMA 传输 UDMASetupTransfer()和启动完成软件传输 StartAndCompleteSWTransfer()等，传输完成后对传输后的数据进行验证。

（1）设置μDMA 传输 UDMASetupTransfer()。设置μDMA 传输 UDMASetupTransfer()的调用语句为

```
UDMASetupTransfer(UDMA_CH0_SW, UDMA_MODE_AUTO, MEM_BUFFER_SIZE,
                  UDMA_SIZE_32, UDMA_ARB_8, g_ulSrcBuf, UDMA_SRC_INC_32,
                  g_ulDstBuf, UDMA_DST_INC_32);
```

设置μDMA 传输 UDMASetupTransfer()的具体内容和 Uart_dma 项目相同。

（2）启动完成软件传输 StartAndCompleteSWTransfer()。启动完成软件传输 StartAndCompleteSWTransfer()的调用语句为

```
StartAndCompleteSWTransfer(UDMA_CH0_SW);
```

启动完成软件传输 StartAndCompleteSWTransfer()的函数格式为

```
unsigned long StartAndCompleteSWTransfer(unsigned long ulChannel);
```

功能：启动完成软件传输。

参数说明：ulChannel，通道选择，传递值是 UDMA_CH0_SW，在 udma.h 中定义为 0x00030000。

返回值：0 表示成功，1 表示错误。

启动完成软件传输 StartAndCompleteSWTransfer()包括下列内容，其中最主要的是请求μDMA 通道 uDMAChannelRequest()开始传输。

- 清除μDMA 中断：uDMAInt Clear()。
- 请求μDMA 通道：uDMAChannelRequest()。
- 获取μDMA 通道模式：uDMAChannel ModeGet()。
- μDMA 通道已允许：uDMAChannelIsEnabled()。

① 清除μDMA 中断 uDMAIntClear()。

```
void uDMAIntClear(unsigned long ulChanMask);
```

功能：清除μDMA 中断。

参数说明：ulChanMask，通道屏蔽，传递值是 0xFFFFFFFF（所有通道）。

uDMAIntClear()的声明在 udma.h 中，代码在 udma.c 中。

② 请求μDMA 通道 uDMAChannelRequest()。

```
void uDMAChannelRequest(unsigned long ulChannelNum);
```

功能：请求μDMA 通道开始传输。

参数说明：ulChannelNum，通道选择，传递值是 0x00030000-通道 0 编码 3（软件）。

uDMAChannelRequest()的声明在 udma.h 中，代码在 udma.c 中。

③ 获取μDMA 通道模式 uDMAChannelModeGet()。

```
unsigned long uDMAChannelModeGet(unsigned long ulChannelStructIndex);
```

功能：获取μDMA 通道模式。

参数说明：ulChannelStructIndex，通道选择，传递值是 0x00030000-通道 0 编码 3（软件）。

返回值：通道模式，0 表示 UDMA_MODE_STOP（停止模式，传输完成）。

uDMAChannelTransferSet()的声明在 udma.h 中，代码在 udma.c 中。

④ μDMA 通道已允许 uDMAChannelIsEnabled()。

```
tBoolean uDMAChannelIsEnabled(unsigned long ulChannelNum);
```

功能：μDMA 通道已允许。

参数说明：ulChannelNum，通道选择，传递值是 0x00030000-通道 0 编码 3（软件）。

返回值：true 表示通道允许，false 表示通道禁止（传输完成）。

uDMAChannelIsEnabled()的声明在 udma.h 中，代码在 udma.c 中。

3．分集μDMA 传输 InitSGTransfer()

分集μDMA 传输 InitSGTransfer()包括下列内容，其中最主要的是分配μDMA 通道 uDMAChannelAssign()和请求μDMA 通道 uDMAChannelRequest()。

- 分配μDMA 通道：uDMAChannelAssign()。
- 禁止μDMA 通道属性：uDMAChannelAttributeDisable()。
- 设置存储器分集传输：SetupSGMemTransfer()。
- 请求μDMA 通道：uDMAChannelRequest()。

（1）分配μDMA 通道 uDMAChannelAssign()。分配μDMA 通道 uDMAChannelAssign()的调用语句为

```
MAP_uDMAChannelAssign(UDMA_CH0_SW);
```

分配μDMA 通道 uDMAChannelAssign()的函数格式为

```
void uDMAChannelAssign(unsigned long ulMapping);
```

功能：分配μDMA 通道。

参数说明：ulMapping，通道选择，传递值是 UDMA_CH0_SW，在 udma.h 中定义为 0x00030000。

uDMAChannelAssign()的声明在 udma.h 中，代码在 udma.c 中。

（2）禁止μDMA 通道属性 uDMAChannelAttributeDisable()。禁止μDMA 通道属性 uDMAChannelAttributeDisable()的调用语句为

```
MAP_uDMAChannelAttributeDisable(UDMA_CH0_SW,UDMA_ATTR_ALTSELECT);
```

禁止μDMA 通道属性 uDMAChannelAttributeDisable()的函数格式为

```
void uDMAChannelAttributeDisable(unsigned long ulChannelNum,
                                 unsigned long ulAttr);
```

功能：禁止μDMA 通道属性。

参数说明：

- ulChannelNum：通道选择，传递值是 UDMA_CH0_SW（0x00030000）。
- ulAttr：通道属性，传递值是 UDMA_ATTR_ ALTSELECT（备选模式）。

uDMAChannelAttributeDisable()的声明在 udma.h 中，代码在 udma.c 中。

（3）设置存储器分集传输 SetupSGMemTransfer()。设置存储器分集传输 SetupSGMemTransfer()的调用语句为

```
SetupSGMemTransfer(UDMA_CH0_SW, pvSrcBuf, pvSrcBuf+BUFF_SIZE/2,
                   pvDstBuf, BUFF_SIZE/2);
```

设置存储器分集传输 SetupSGMemTransfer()的函数格式为

```
static void SetupSGMemTransfer(unsigned long ulChannel, void *pvSrcBuf0,
                               void *pvSrcBuf1, void *pvDstBuf,
                               unsigned long size);
```

功能：设置存储器分集传输。

参数说明：

- ulChannel：通道选择，传递值是 UDMA_CH0_SW，在 udma.h 中定义为 0x00030000。
- *pvSrcBuf0：任务 1 源缓存指针，传递值是 pvSrcBuf。
- *pvSrcBuf1：任务 2 源缓存指针，传递值是 pvSrcBuf+BUFF_SIZE/2。
- *pvDstBuf：目的缓存指针，传递值是 pvDstBuf。
- size：传输项数，传递值是 BUFF_SIZE/2。

设置存储器分集传输 SetupSGMemTransfer()通过创建分集任务表宏定义 uDMATaskStructEntry()创建任务表 0 和任务表 1，通过调用设置μDMA 分集传输 uDMAChannelScatterGatherSet()，以及允许μDMA 通道 uDMAChannelEnable()实现存储器分集传输的设置。

① 创建分集任务表宏定义 uDMATaskStructEntry()。创建分集任务表宏定义 uDMATaskStructEntry()的调用语句为

```
tDMAControlTable task0=uDMATaskStructEntry(size, UDMA_SIZE_8,
                    UDMA_SRC_INC_8, pvSrcBuf0, UDMA_DST_INC_8, pvDstBuf,
                    UDMA_ARB_8, UDMA_MODE_MEM_SCATTER_GATHER);

tDMAControlTable task1=uDMATaskStructEntry(size, UDMA_SIZE_8,
                    UDMA_SRC_INC_8, pvSrcBuf1, UDMA_DST_INC_8,
                   (char *)pvDstBuf+size, UDMA_ARB_8, UDMA_MODE_AUTO);
```

创建分集任务表宏定义 uDMATaskStructEntry()的格式为

```
uDMATaskStructEntry(ulTransferCount, ulItemSize, ulSrcIncrement,
                    pvSrcAddr,ulDstIncrement, pvDstAddr, ulArbSize, ulMode)
```

功能：创建分集任务表。

参数说明：

- ulTransferCount：传输项数，传递值是 size（10）。
- ulItemSize：传输项大小，传递值是 UDMA_SIZE_8（8 位）。
- ulSrcIncrement：源增量，传递值是 UDMA_SRC_INC_8（8 位）。
- pvSrcAddr：源缓存地址指针，分别指向 pvSrcBuf0 和 pvSrcBuf1。
- ulDstIncrement：目的增量，传递值是 UDMA_DST_INC_8（8 位）。
- pvDstAddr：目的缓存地址指针，分别指向 pvDstBuf 和 pvDstBuf+size。
- ulArbSize：仲裁大小，传递值是 UDMA_ARB_8（8 项）。
- ulMode：通道模式，传递值分别是 UDMA_MODE_MEM_SCATTER_GATHER（存储器分集模式）和 UDMA_MODE_AUTO（自动模式）。

uDMATaskStructEntry()的定义在 udma.h。

② 设置μDMA 分集传输 uDMAChannelScatterGatherSet()。设置μDMA 分集传输 uDMAChannelScatterGatherSet()的调用语句为

```
uDMAChannelScatterGatherSet(ulChannel, 2, TaskList, 0);
```

设置μDMA 分集传输 uDMAChannelScatterGatherSet()的函数格式为

```
void uDMAChannelScatterGatherSet(unsigned long ulChannelNum,
                                 unsigned ulTaskCount, void *pvTaskList,
                                 unsigned long ulIsPeriphSG);
```

功能：设置μDMA 分集传输。

参数说明：

- ulChannelNum：通道选择，传递值是 0x00030000，表示通道 0 编码 3（软件）。

- ulTaskCount：任务数，传递值是 2。
- pvTaskList：任务表，传递值是任务表 0+任务表 1。
- ulIsPeriphSG：分集标志（0 表示存储器分集，1 表示外设分集），传递值是 0（存储器分集）。

uDMAChannelScatterGatherSet()的声明在 udma.h 中，代码在 udma.c 中。

③ 允许μDMA 通道 uDMAChannelEnable()。

```
void uDMAChannelEnable(unsigned long ulChannelNum);
```

功能：允许μDMA 通道。

参数说明：ulChannelNum，通道选择，传递值是 0x00030000，表示通道 0 编码 3（软件）。

uDMAChannelEnable()的声明在 udma.h 中，代码在 udma.c 中。

（4）请求μDMA 通道 uDMAChannelRequest()。

```
void uDMAChannelRequest(unsigned long ulChannelNum);
```

功能：请求μDMA 通道开始传输。

参数说明：ulChannelNum，通道选择，传递值是 0x00030000，表示通道 0 编码 3（软件）。

uDMAChannelRequest()的声明在 udma.h 中，代码在 udma.c 中。

请求μDMA 通道完成传输后，系统产生μDMA 软件中断，调用μDMA 软件中断处理程序 DmaSwIntHandler()：设置完成标志（iDone = 1），获取μDMA 中断状态 uDMAIntStatus()，清除μDMA 中断 uDMAIntClear()。

```
unsigned long uDMAIntStatus(void);
```

功能：获取μDMA 中断状态。

返回值：产生中断的通道位屏蔽，1 表示通道 0。

uDMAIntStatus()的声明在 udma.h 中，代码在 udma.c 中。

```
void uDMAIntClear(unsigned long ulChanMask);
```

功能：清除μDMA 中断。

参数说明：ulChanMask，通道位屏蔽，传递值是 1（通道 0）。

uDMAIntClear()的声明在 udma.h 中，代码在 udma.c 中。

InitSGTransfer()通过判断 iDone 检查传输是否完成，没完成等待，完成后通过比较传输前后的数据是否一致检查传输是否成功。

4．乒乓 UART 传输

乒乓 UART 传输包括初始化 UART 传输 InitUART0Transfer()、UART 中断处理 UART0IntHandler()和计算传输速率等。

（1）初始化 UART 传输 InitUART0Transfer()。初始化 UART 传输 InitUART0Transfer()包括复位 UART、允许 UART 时钟、配置 UART 接口、分配μDMA 通道、注册 UART 中断、设置 UART FIFO 阈值、允许 UART、设置 UART 环回模式、允许 UART 中断、设置传输和允许 UART DMA 等。

① 复位 UART 和允许 UART 时钟。复位 UART 通过调用 PRCMPeripheralReset()实现，允许 UART 时钟通过调用 PRCM PeripheralClkEnable()实现。

② 配置 UART 接口。配置 UART 接口通过调用 UARTConfigSetExpClk()将 UART 配置为：波特率 115200、8 位数据、1 个停止位和无校验。

③ 分配μDMA 通道。分配μDMA 通道通过调用 uDMAChannelAssign()将μDMA 通道 8 设置为 UARTA0_RX（UDMA_CH8_UARTA0_RX），通道 9 设置为 UARTA0_TX（UDMA_CH9_UARTA0_TX）。

④ 注册 UART 中断。注册 UART 中断通过调用 UARTIntRegister()将 UARTA0 中断处理程序设置为 UART0IntHandler。

⑤ 设置 UART FIFO 阈值。设置 UART FIFO 阈值通过调用 UARTFIFOLevelSet()将发送和接收 FIFO 的阈值都设为 4（UART_FIFO_TX4_8 和 UART_FIFO_RX4_8）。

⑥ 允许 UART。允许 UART 通过调用 UARTEnable()实现。

⑦ 设置 UART 环回模式。设置 UART 环回模式通过写寄存器 HWREG()实现。

⑧ 允许 UART 中断。允许 UART 中断通过调用 UARTIntEnable()允许 UART DMA 发送中断和接收中断（UART_INT_DMATX 和 UART_INT_DMARX）。

⑨ 设置μDMA 传输。设置μDMA 传输通过调用 UDMASetupTransfer()将 UDMA_CH8_UARTA0_RX 主选择（UDMA_PRI_SELECT）和备用选择（UDMA_ALT_SELECT），以及 UDMA_CH9_UARTA0_TX 分别设置为表 4.9 所示值。

表 4.9　UDMATransfer()设置值

参　数	UDMA_PRI_SELECT	UDMA_ALT_SELECT	UDMA_CH9_UARTA0_TX
通道模式	乒乓模式	乒乓模式	基本模式
传输项数	sizeof(g_ucRxBufA)	sizeof(g_ucRxBufB)	sizeof(g_ucTxBuf)
传输项大小	8 位	8 位	8 位
仲裁大小	4 项	4 项	4 项
源缓存	UART 数据寄存器	UART 数据寄存器	g_ucTxBuf
源增量	无增量	无增量	8 位
目的缓存	g_ucRxBufA	g_ucRxBufB	UART 数据寄存器
目的增量	8 位	8 位	无增量

⑩ 允许 UART DMA。允许 UART DMA 通过调用 UARTDMAEnable()允许 UART DMA 发送和接收。

（2）UART 中断处理 UART0IntHandler()。初始化 UART 传输后通过中断方式实现 UART DMA 传输，UART 中断处理主要包括获取 UART 中断状态、清除 UART 中断、获取μDMA 通道模式和μDMA 通道已允许等。

① 获取 UART 中断状态。获取 UART 中断状态通过调用 UARTIntStatus()实现，返回的中断状态依次为 131072（0x0002 0000），即 UART_INT_DMATX（DMA 发送中断），以及 196608（0x0003 0000），即 UART_INT_DMARX（DMA 接收中断）。

② 清除 UART 中断。清除 UART 中断通过调用 UARTIntClear()清除 DMA 发送或接收中断。

③ 获取μDMA 通道模式。获取μDMA 通道模式通过调用 uDMAChannelModeGet()实现，返回值依次为 3（UDMA_ MODE_PINGPONG，乒乓模式）和 0（UDMA_MODE_STOP，停止模式）。

④ μDMA 通道已允许。μDMA 通道已允许通过调用 uDMAChannelIsEnabled()实现，返回值是 false，即通道禁止（传输完成）。

UART 中断处理程序 UART0IntHandler()通过 DMA 通道 9 发送 g_ucTxBuf 中的数据，数据显示在超级终端中，并环回到接收端，通过 DMA 通道 8 主选择和备用选择乒乓接收到 g_ucRxBufA 和 g_ucRxBufB 中。传输 6 s 后停止传输，设置完成标志（UARTDone = 1），注销 UART 中断。

（3）计算传输速率。程序将每秒种的传输数据数存放在数组 ulBytesTransferred[]中，6 s 传输完成后将后 5 s 的数据数平均，并显示在超级终端中。

综上所述，udma 项目使用的与μDMA 有关的 API 函数如表 4.10 所示。

表 4.10 udma 项目使用的与μDMA 有关的 API 函数

功 能	函数名	调用函数	代码文件
允许μDMA 时钟	**PRCMPeripheralClkEnable()**	UDMAInit()	prcm.c
复位μDMA	PRCMPeripheralReset()	UDMAInit()	prcm.c
注册μDMA 中断	uDMAIntRegister()	UDMAInit()	udma.c
允许μDMA	**uDMAEnable()**	UDMAInit()	udma.c
设置μDMA 控制表	uDMAControlBaseSet()	UDMAInit()	udma.c
设置μDMA 传输	**UDMASetupTransfer()**	InitSWTransfer() 、 InitUART0Transfer() 、 UART0IntHandler()	udma_if.c
设置μDMA 通道控制	uDMAChannelControlSet()	UDMASetupTransfer()	udma.c
允许μDMA 通道属性	uDMAChannelAttributeEnable()	UDMASetupTransfer()	udma.c
设置μDMA 通道传输	uDMAChannelTransferSet()	UDMASetupTransfer()	udma.c
允许μDMA 通道	uDMAChannelEnable()	UDMASetupTransfer()、SetupSGMemTransfer()、UART0IntHandler()	udma.c
清除μDMA 中断	uDMAIntClear()	StartAndCompleteSWTransfer()、DmaSwIntHandler()	udma.c
请求μDMA 通道	**uDMAChannelRequest()**	StartAndCompleteSWTransfer()、InitSGTransfer()	udma.c
获取μDMA 通道模式	uDMAChannelModeGet()	StartAndCompleteSWTransfer()、UART0IntHandler()	udma.c
μDMA 通道已允许	uDMAChannelIsEnabled()	StartAndCompleteSWTransfer()、UART0IntHandler()	udma.c
分配μDMA 通道	**uDMAChannelAssign()**	InitSGTransfer()、InitUART0Transfer()	udma.c
禁止μDMA 通道属性	uDMAChannelAttributeDisable()	InitSGTransfer()	udma.c
设置μDMA 分集传输	uDMAChannelScatterGatherSet()	SetupSGMemTransfer()	udma.c
获取μDMA 中断状态	uDMAIntStatus()	DmaSwIntHandler()	udma.c

第 5 章

网络应用入门

CC3200 网络驱动 API 包括下列 6 部分。

- Device API：提供硬件相关管理功能，如启动、停止、设置和读取设备配置等。
- WLAN API：提供 WLAN 和 802.11 协议相关管理功能，如设置设备模式（站点、接入点或直连）、设置配置方法、增加连接配置文件和设置连接策略等。
- Socket API：提供套接字相关管理功能，包括用户应用程序常用的 API。
- NetApp API：提供不同的网络服务功能，包括 HTTP 服务器服务、DHCP 服务器服务和 MDNS 客户机/服务器服务等。
- NetCfg API：提供不同的网络参数配置功能，如设置 MAC 地址、通过 DHCP 获取 IP 地址和设置静态 IP 地址等。
- File System API：提供串行闪存访问功能，如网络或用户数据的读写操作等。

使用 CC3200 进行网络编程需要下列软件模块。

- Wi-Fi 子系统初始化：从休眠状态唤醒 Wi-Fi 子系统，包括下列主要步骤。
 - ✧ 允许 LSPI 接口。
 - ✧ 注册异步事件处理程序。
 - ✧ 允许 Wi-Fi 子系统。
 - ✧ 向 Wi-Fi 子系统发送同步信息并等待初始化完成后返回的中断请求信号。
- 网络配置：主要是 Wi-Fi 子系统角色配置，例如从站点切换到接入点。为了使新配置生效，必须重新启动 Wi-Fi 子系统。
- WLAN 连接：建立 WLAN 通信物理接口，例如，作为站点人工连接到接入点。在使用 TCP/UDP 套接字之前，必须获得 IP 地址。
- 套接字连接：设置 TCP/UDP，包括下列内容。
 - ✧ 创建套接字：选择 TCP、UDP 或 RAW 套接字，选择客户端或服务器套接字，定义阻塞/非阻塞和超时等套接字特性。
 - ✧ 查询服务器 IP 地址：实现客户端通信时，客户端不知道建立套接字连接所需要的远端服务器 IP 地址，使用 DNS 协议可以用服务器名称查询服务器 IP 地址。
 - ✧ 创建套接字连接：对于 TCP 套接字，需要在数据传输前建立套接字连接。
- 数据传输：套接字连接建立后，就可以在客户端和服务器间实现应用程序要求的数据传输。
- 断开套接字：完成数据传输后，关闭套接字通信通道。

- Wi-Fi 子系统休眠：当 Wi-Fi 子系统长时间不工作时，进入休眠。

基本的网络应用程序状态机如图 5.1 所示，本章主要分析两个网络应用入门示例程序：wlan_station 示例程序和 wlan_ap 示例程序，其中 wlan_station 示例程序可以作为编写网络应用新程序的基础。

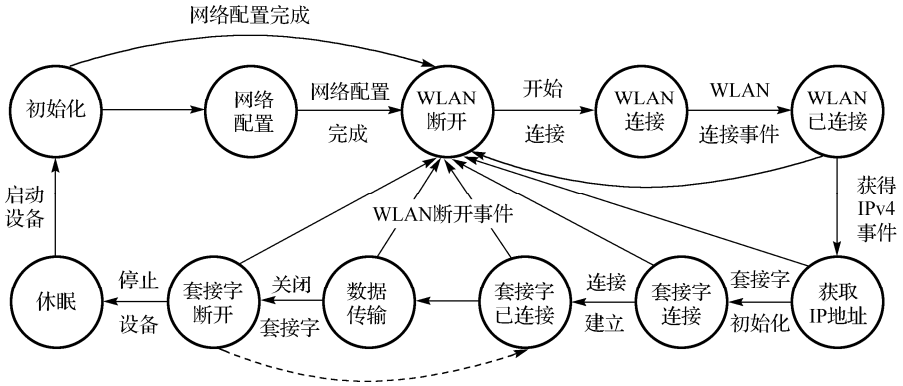


图 5.1　基本的网络应用程序状态机

5.1　Wlan_station 示例程序分析

Wlan_station 示例程序在“example\getting_started_with_wlan_station”目录中，Wlan_station 项目的工作示意图如图 5.2 所示。

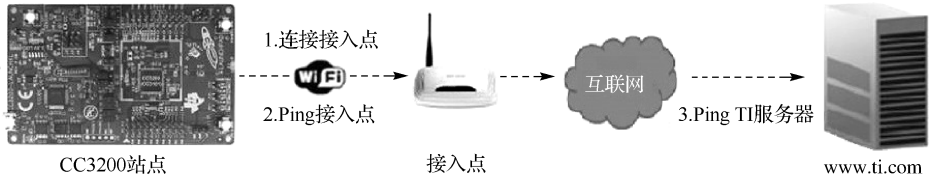


图 5.2　Wlan_station 项目工作示意图

CC3200 站点首先连接接入点，然后分别 Ping 接入点和 TI 服务器。

Wlan_station 项目包含下列 4 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO 和 UART 引脚配置程序文件。
- gpio_if.c：GPIO 接口驱动程序文件。
- uart_if.c：UART 接口驱动程序文件。

Wlan_station 项目程序流程如图 5.3 所示。

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（2）在 PC 上打开超级终端 Tera Term。

（3）将“example\common\common.h”中的下列宏定义。

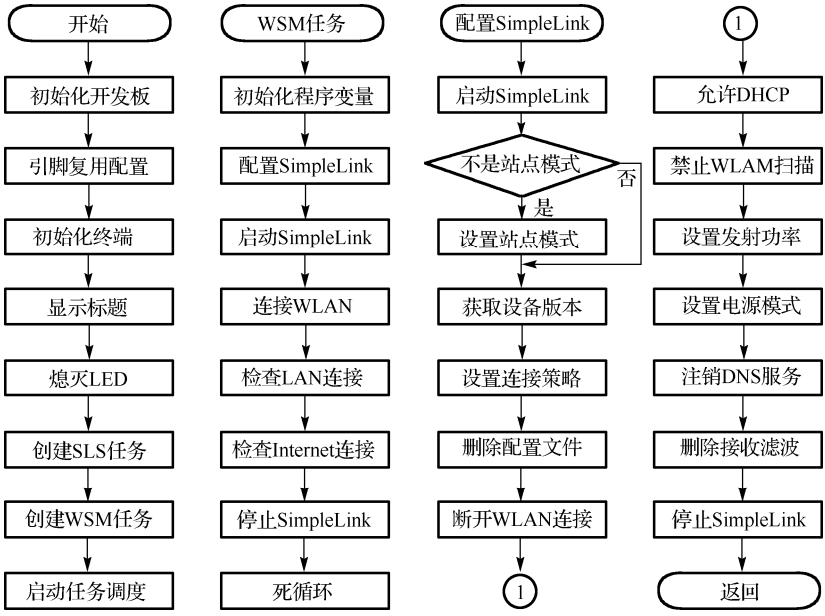


图 5.3　Wlan_station 项目程序流程图

```
#define SSID_NAME       "cc3200demo"        /* 接入点名称 */
#define SECURITY_TYPE   SL_SEC_TYPE_OPEN    /* 安全类型(OPEN/WEP/WPA) */
#define SECURITY_KEY    ""                  /* 安全密码 */
```

修改为自己 AP 的配置值，例如：

```
#define SSID_NAME       "1234"              /* 接入点名称 */
#define SECURITY_TYPE   SL_SEC_TYPE_WPA     /* 安全类型(OPEN/WEP/WPA) */
#define SECURITY_KEY    "1234asdf"          /* 安全密码 */
```

（4）在 CCS Project Explorer（项目浏览器）中右击 wlan_station 项目，在弹出菜单中选择“Build Project”生成目标文件 wlan_station.out。

注意：wlan_station 项目使用了 ti_rtos_config 项目，因此必须在项目浏览器中导入 ti_rtos_config 项目，并参考 2.1.2 节中重建 SDK 驱动程序库重建 ti_rtos_config 项目。

（5）在 CCS Project Explorer（项目浏览器）中右击 wlan_station 项目，在弹出菜单中选择“Debug As”→“Code Composer Debug Session”，将 wlan_station.out 下载到开发板的 CC3200 中，然后进入调试界面。

（6）单击调试工具栏中的“Resume”（运行）按钮运行程序，开发板上的 3 个 LED 首先全部熄灭，然后红 LED 闪烁，最后红 LED、绿 LED 和黄 LED 依次点亮。

超级终端中显示信息如图 5.4 所示。

（7）单击调试工具栏中的“Terminate”（结束）按钮，结束调试返回编辑界面。

（8）重复（5）重新进入调试界面，为程序的分析做准备。

下面详细介绍程序的分析步骤，包括主函数 main()分析和 WLAN 站点模式（WSM）函数 WlanStationMode()分析。

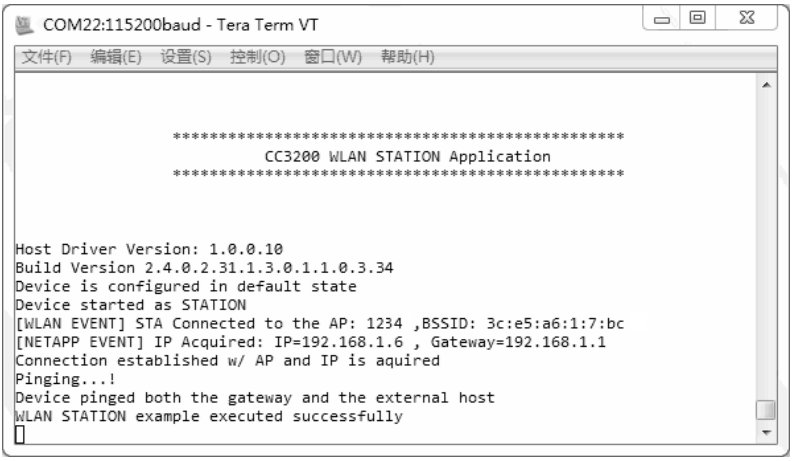


图 5.4　Wlan_station 项目运行结果

5.1.1　主函数 main()分析

主函数 main()包括初始化开发板、引脚复用配置、初始化终端、显示标题、熄灭 LED、创建 SLS（SimpleLinkSpawn）任务 VStartSimpleLinkSpawnTask()、创建 WSM（WlanStation Mode）任务 osi_TaskCreate()和启动任务调度 osi_start()等。

初始化开发板、引脚复用配置、初始化终端、显示标题和熄灭 LED 比较简单，可以一次通过，方法是右击 main()中的语句

```
lRetVal = VStartSimpleLinkSpawnTask(SPAWN_TASK_PRIORITY);
```

在弹出菜单中单击“Run to Line”（运行到行），依次运行初始化开发板、引脚复用配置、初始化终端、显示标题和熄灭 LED。

显示标题在超级终端中显示下列标题。

```
*******************************************************************************
                    CC3200 WLAN STATION Application
*******************************************************************************
```

熄灭 LED 将开发板上的 3 个 LED 熄灭。

1．创建 SLS 任务 VStartSimpleLinkSpawnTask()

创建 SLS 任务的函数是 VStartSimpleLinkSpawnTask()，实际参数是 SPAWN_TASK_PRIORITY，在 main.c 中定义为 9。

（1）单击调试工具栏中的“Step Into”（单步进入）按钮，进入 VStartSimpleLinkSpawnTask()，函数格式为

```
OsiReturnVal_e VStartSimpleLinkSpawnTask(
                                        unsigned portBASE_TYPE uxPriority);
```

功能：创建 SLS 任务。

参数说明：uxPriority，任务优先级，传递值是 9（高优先级）。

返回值：OSI_OK（0）表示成功，OSI_FAILURE（–1）表示失败。

VStartSimpleLinkSpawnTask()的声明在 osi.h 中，代码在 osi_tirtos.c 中。VStartSimpleLinkSpawnTask()通过调用 Mailbox_create()和 Task_create()完成 SLS 任务的创建。

（2）单击调试工具栏中的“Step Return”（单步返回）按钮，退出 VStartSimpleLinkSpawnTask()，回到 main()函数。

（3）单击调试工具栏中的“Step Over”（单步跨越）按钮，lRetVal 的值由–1 变为 0。如果 VStartSimpleLinkSpawnTask()操作失败（lRetVal < 0），程序将在超级终端中显示错误代码等信息，然后进入死循环。如果 VStartSimpleLinkSpawnTask()操作成功（lRetVal = 0），程序将运行下列语句。

```
lRetVal = osi_TaskCreate(WlanStationMode,
                         (const signed char*)"Wlan Station Task",
                          OSI_STACK_SIZE, NULL, 1, NULL);
```

2．创建 WSM 任务 osi_TaskCreate()

创建 WSM 任务的函数是 osi_TaskCreate()，共有 6 个实际参数，依次是 WlanStationMode、“Wlan Station Task”、OSI_STACK_SIZE、NULL、1 和 NULL，其中 OSI_STACK_SIZE 在 main.c 中定义为 2048。

（1）单击调试工具栏中的“Step Into”（单步进入）按钮两次，进入 osi_TaskCreate()，函数格式为

```
OsiReturnVal_e osi_TaskCreate(P_OSI_TASK_ENTRY pEntry,
            const signed char * const pcName, unsigned short usStackDepth,
            void *pvParameters, unsigned long uxPriority,
            OsiTaskHandle* pTaskHandle);
```

功能：创建任务。

参数说明：

- pEntry：任务函数指针，WSM 任务函数是 WlanStationMode。
- pcName：任务名称指针，WSM 任务名称是 Wlan Station Task。
- usStackDepth：任务堆栈深度，传递值是 2048（OSI_STACK_SIZE）。
- pvParameters：任务参数指针，传递值是 NULL（空）。
- uxPriority：任务优先级，传递值是 1（低优先级）。
- pTaskHandle：任务句柄指针，传递值是 NULL（空）。

返回值：OSI_OK（0）表示成功，OSI_FAILURE（–1）表示失败。

osi_TaskCreate()的声明在 osi.h 中，代码在 osi_tirtos.c 中。osi_TaskCreate()通过调用 Task_create()完成任务的创建。

（2）单击调试工具栏中的“Step Return”（单步返回）按钮，退出 osi_TaskCreate()，回到 main()函数。

（3）单击调试工具栏中的“Step Over”（单步跨越）按钮，lRetVal 的值保持 0。如果

osi_TaskCreate()操作失败（lRetVal < 0），程序将在超级终端中显示错误代码等信息，然后进入死循环。如果 osi_TaskCreate()操作成功（lRetVal = 0），程序将运行下列语句。

```
osi_start();
```

为了实现 WSM 任务 WlanStationMode 的分析，在运行 osi_start()前，需先在 WlanStationMode()的下列语句设置断点。

```
InitializeAppVariables();
```

3．启动任务调度 osi_start()

启动任务调度的函数是 osi_start()。

（1）单击调试工具栏中的“Step Into”（单步进入）按钮两次，进入 osi_start()，函数格式为

```
void osi_start();
```

功能：启动任务调度。

osi_ start()的声明在 osi.h 中，代码在 osi_tirtos.c 中。osi_ start()通过调用 BIOS_start()完成任务调度的启动。

（2）单击调试工具栏中的“Step Return”（单步返回）按钮，运行 BIOS_start()启动任务调度，运行 WlanStationMode()，程序停止 WlanStationMode()中的断点处。

5.1.2 WlanStationMode()分析

WlanStationMode()是 wlan_station 项目的核心函数，主要包括下列内容，其中最主要的是 sl_Start()、sl_WlanConnect()和 sl_Stop()。

- 初始化程序变量：InitializeAppVariables()。
- 配置 SimpleLink 为默认状态：ConfigureSimpleLinkToDefaultState()。
- 启动 SimpleLink：sl_Start()。
- 连接 WLAN：WlanConnect()（sl_WlanConnect()）。
- 检查 LAN 连接：CheckLanConnection()。
- 检查 Internet 连接：CheckInternetConnection()。
- 停止 SimpleLink：sl_Stop()。

1．初始化程序变量 InitializeAppVariables()

初始化程序变量的函数是 InitializeAppVariables()。

（1）单击调试工具栏中的“Step Into”（单步进入）按钮，进入 InitializeAppVariables()，函数格式为

```
static void InitializeAppVariables();
```

功能：初始化程序变量。

初始化程序变量主要对下列全局变量进行初始化。

- g_ulStatus：设备状态，0。
- g_ulPingPacketsRecv：接收数据包数，0。
- g_ulGatewayIP：网关 IP 地址，0。
- g_ucConnectionSSID：接入点名称，0。
- g_ucConnectionBSSID：接入点 MAC 地址，0。

（2）单击调试工具栏中的“Step Return”（单步返回）按钮，退出 InitializeAppVariables()，回到 WlanStationMode()中的下列语句。

```
lRetVal = ConfigureSimpleLinkToDefaultState();
```

2．配置 SimpleLink 为默认状态 ConfigureSimpleLinkToDefaultState()

配置 SimpleLink 为默认状态的函数是 ConfigureSimpleLinkToDefaultState()。

单击调试工具栏中的“Step Into”（单步进入）按钮，进入 ConfigureSimpleLinkToDefaultState()，函数格式为

```
static long ConfigureSimpleLinkToDefaultState();
```

功能：配置 SimpleLink 为默认状态。

返回值：0 表示成功，负值表示错误。

ConfigureSimpleLinkToDefaultState()主要包括下列内容。

- 启动 SimpleLink：sl_Start()。
- 获取设备版本：sl_DevGet()。
- 设置连接策略：sl_WlanPolicySet()。
- 删除配置文件：sl_WlanProfileDel()。
- 断开 WLAN 连接：sl_WlanDisconnect()。
- 允许 DHCP：sl_NetCfgSet()。
- 禁止 WLAN 扫描：sl_WlanPolicySet()。
- 设置发射功率：sl_WlanSet()。
- 设置电源模式：sl_WlanPolicySet()。
- 注销 DNS 服务：sl_NetAppMDNSUnRegisterService()。
- 删除接收滤波：sl_WlanRxFilterSet()。
- 停止 SimpleLink：sl_Stop()。

对于初学者，可以忽略 ConfigureSimpleLinkToDefaultState()的分析过程，单击调试工具栏中的“Step Return”（单步返回）按钮，退出 ConfigureSimpleLinkToDefaultState()，返回到 WlanStationMode()，然后转到“3．启动 SimpleLink sl_Start()”继续分析。

为了简化说明，以下操作只介绍功能，不再介绍操作步骤。

（1）启动 SimpleLink 函数 sl_Start()。sl_Start()的调用语句为

```
lMode = sl_Start(0, 0, 0);
```

sl_Start()的函数格式为

```
_i16 sl_Start(const void* pIfHdl, _i8* pDevName,
              const P_INIT_CALLBACK pInitCallBack);
```

功能：启动 SimpleLink。

参数说明：

- pIfHdl：处理函数指针，传递值是 0。
- pDevName：设备名指针，传递值是 0。
- pInitCallBack：回调函数指针，传递值是 0。

返回值：当前角色或错误代码，包括下列值。

- 0 表示站点模式（ROLE_STA）。
- 2 表示接入点模式（ROLE_AP）。
- 3 表示点对点模式（ROLE_ P2P）。
- 负值表示错误。

sl_Start()的声明在 device.h 中，代码在 device.c 中。sl_Start()的返回值 lMode 是 0，表示启动 SimpleLink 进入站点模式。

如果 SimpleLink 不是站点模式，则调用 sl_WlanSetMode()设置站点模式，并通过停止 SimpleLink 和启动 SimpleLink 重启 SimpleLink。ASSERT_ON_ERROR()用于检查返回值，如果返回值为负值（错误），则调用 ERR_PRINT()即 Report()在超级终端中显示错误代码等信息。ASSERT_ON_ERROR()和 ERR_PRINT()都在 common.h 中定义。

（2）获取设备版本 sl_DevGet()。sl_DevGet()的调用语句为

```
lRetVal = sl_DevGet(SL_DEVICE_GENERAL_CONFIGURATION, &ucConfigOpt,
                    &ucConfigLen, (unsigned char *)(&ver));
```

sl_DevGet()的函数格式为

```
_i32 sl_DevGet(_u8 DeviceGetId, _u8 *pOption, _u8 *pConfigLen, _u8 *pValues);
```

功能：获取设备配置。

参数说明：

- DeviceGetId：配置 ID，传递值是 SL_DEVICE_GENERAL_CONFIGURATION（设备通用配置），在 device.c 中定义为 1。
- pOption：配置选项指针，传递变量名 ucConfigOpt，传递值是 SL_DEVICE_GENERAL_VERSION（设备通用版本），在 device.c 中定义为 12。
- pConfigLen：配置值长度指针，传递变量名是 ucConfigLen，传递值是 44。
- pValues：配置值指针，用于存放获取的配置值，传递变量名是 ver。

返回值：0 表示成功，负值表示错误。

sl_DevGet()的声明在 device.h 中，代码在 device.c 中。sl_DevGet()的返回值 lRetVal 是 0：获取设备配置成功。获取设备配置成功后，程序调用 UART_PRINT()在超级终端中显示设备配置如下。

```
Host Driver Version: 1.0.0.10
```

主机驱动程序版本：1.0.0.10，即 SDK 版本。

```
Build Version 2.4.0.2.31.1.3.0.1.1.0.3.34
```

生成版本 2.4.0.2.31.1.3.0.1.1.0.3.34，其中 2.4.0.2 是 NWP 版本，1.3.0.1 是 MAC 版本，1.0.3.34 是 PHY 版本。

（3）设置连接策略 sl_WlanPolicySet()。sl_WlanPolicySet()的调用语句为

```
lRetVal = sl_WlanPolicySet(SL_POLICY_CONNECTION,
                           SL_CONNECTION_POLICY(1, 0, 0, 0, 1), NULL, 0);
```

sl_WlanPolicySet()的函数格式为

```
_i16 sl_WlanPolicySet(_u8 Type, const _u8 Policy, _u8 *pVal, _u8 ValLen);
```

功能：设置 WLAN 策略。

参数说明：

- Type：策略类型，传递值是 SL_POLICY_CONNECTION（连接），在 wlan.h 中定义为 0x10。
- Policy：策略，传递值是 SL_CONNECTION_POLICY(1, 0, 0, 0, 1)（连接策略），在 wlan.h 中定义为 SL_CONNECTION_POLICY(Auto, Fast, Open, anyP2P, autoSmartConfig)，Auto=1 表示自动连接，autoSmartConfig=1 表示自动智能配置连接。
- pVal：返回值指针，传递值是 NULL（空）。
- ValLen：返回值长度，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_WlanPolicySet()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanPolicySet()的返回值 lRetVal 是 0，表示设置 WLAN 策略成功。

（4）删除配置文件 sl_WlanProfileDel()。sl_WlanProfileDel()的调用语句为

```
lRetVal = sl_WlanProfileDel(0xFF);
```

sl_WlanProfileDel()的函数格式为

```
_i16 sl_WlanProfileDel(_i16 Index);
```

功能：删除 WLAN 配置文件。

参数说明：Index，配置文件号，传递值是 0xFF（所有配置文件）。

返回值：0 表示成功，负值表示错误。

sl_WlanProfileDel()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanProfileDel()的返回值 lRetVal 是 0，表示删除 WLAN 配置文件成功。

（5）断开 WLAN 连接 sl_WlanDisconnect()。sl_WlanDisconnect()的调用语句为

```
lRetVal = sl_WlanDisconnect();
```

sl_WlanDisconnect()的函数格式为

```
_i16 sl_WlanDisconnect(void);
```

功能：断开 WLAN 连接。

返回值：0 表示断开 WLAN 连接，其他值表示 WLAN 连接已经断开。

sl_WlanDisconnect()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanDisconnect()的返回值 lRetVal 是−129，表示 WLAN 连接已经断开。

（6）允许 DHCP 函数 sl_NetCfgSet()。sl_NetCfgSet()的调用语句为

```
lRetVal = sl_NetCfgSet(SL_IPV4_STA_P2P_CL_DHCP_ENABLE,1,1,&ucVal);
```

sl_NetCfgSet()的函数格式为

```
_i32 sl_NetCfgSet(_u8 ConfigId, _u8 ConfigOpt, _u8 ConfigLen, _u8 *pValues);
```

功能：设置网络配置。

参数说明：

- ConfigId：配置 ID，传递值是 SL_IPV4_STA_P2P_CL_DHCP_ENABLE（允许 DHCP），在 netcfg.h 中定义为 4。
- ConfigOpt：配置选项，传递值是 1。
- ConfigLen：配置长度，传递值是 1。
- *pValues：配置值指针，传递变量名是 ucVal，传递值是 1。

返回值：0 表示成功，负值表示错误。

sl_NetCfgSet()的声明在 netcfg.h 中，代码在 netcfg.c 中。sl_NetCfgSet()的返回值 lRetVal 是 0，表示设置网络配置成功。

（7）禁止 WLAN 扫描 sl_WlanPolicySet()。sl_WlanPolicySet()的调用语句为

```
lRetVal = sl_WlanPolicySet(SL_POLICY_SCAN, ucConfigOpt, NULL, 0);
```

sl_WlanPolicySet()的函数格式为

```
_i16 sl_WlanPolicySet(_u8 Type, const _u8 Policy, _u8 *pVal, _u8 ValLen);
```

功能：设置 WLAN 策略。

参数说明：

- Type：策略类型，传递值是 SL_POLICY_SCAN（扫描），在 wlan.h 中定义为 0x20。
- Policy：策略，传递变量名是 ucConfigOpt，传递值是 SL_SCAN_POLICY(0)（扫描策略），在 wlan.h 中定义为 SL_SCAN_ POLICY(Enable)，Enable=0 表示禁止扫描。
- pVal：返回值指针，传递值是 NULL（空）。
- ValLen：返回值长度，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_WlanPolicySet()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanPolicySet()的返回值 lRetVal 是 0：设置 WLAN 策略成功。

（8）设置发射功率 sl_WlanSet()。sl_WlanSet()的调用语句为

```
lRetVal = sl_WlanSet(SL_WLAN_CFG_GENERAL_PARAM_ID,
                     WLAN_GENERAL_PARAM_OPT_STA_TX_POWER,
                     1, (unsigned char *)&ucPower);
```

sl_WlanSet()的函数格式为

```
_i16 sl_WlanSet(_u16 ConfigId, _u16 ConfigOpt, _u16 ConfigLen, _u8 *pValues);
```

功能：设置 WLAN 配置。

参数说明：

- ConfigId：配置 ID，传递值是 SL_WLAN_CFG_GENERAL_PARAM_ID（参数 ID），在 wlan.h 中定义为 1。
- ConfigOpt：配置选项，传递值是 WLAN_GENERAL_PARAM_OPT_STA_TX_POWER（站点发射功率），在 wlan.h 中定义为 10。
- ConfigLen：配置长度，传递值是 1。
- pValues：配置值指针，传递变量名是 ucPower，传递值是 0～15，0 表示最大功率。

返回值：0 表示成功，负值表示错误。

sl_WlanSet()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanSet()的返回值 lRetVal 是 0：设置 WLAN 配置成功。

（9）设置电源模式 sl_WlanPolicySet()。sl_WlanPolicySet()的调用语句为

```
lRetVal = sl_WlanPolicySet(SL_POLICY_PM, SL_NORMAL_POLICY, NULL, 0);
```

sl_WlanPolicySet()的函数格式为

```
_i16 sl_WlanPolicySet(_u8 Type, const _u8 Policy, _u8 *pVal, _u8 ValLen);
```

功能：设置 WLAN 策略。

参数说明：

- Type：策略类型，传递值是 SL_POLICY_PM（电源管理），在 wlan.h 中定义为 0x30。
- Policy：策略，传递值是 SL_NORMAL_POLICY（正常策略），在 wlan.h 中为 0。
- pVal：返回值指针，传递值是 NULL（空）。
- ValLen：返回值长度，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_WlanPolicySet()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanPolicySet()的返回值 lRetVal 是 0，表示设置 WLAN 策略成功。

（10）注销 DNS 服务 sl_NetAppMDNSUnRegisterService()。sl_NetAppMDNSUnRegister Service()的调用语句为

```
lRetVal = sl_NetAppMDNSUnRegisterService(0, 0);
```

sl_NetAppMDNSUnRegister Service()的函数格式为

```
_i16 sl_NetAppMDNSUnRegisterService(const _i8* pServiceName,
                                    _u8 ServiceNameLen);
```

功能：注销 DNS 服务。

参数说明：

- pServiceName：服务名指针，传递值是 0。

- ServiceNameLen：服务名长度，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_NetAppMDNSUnRegisterService() 的声明在 netapp.h 中，代码在 netapp.c 中。sl_NetAppMDNSUnRegisterService()的返回值 lRetVal 是 0：注销 DNS 服务成功。

（11）删除接收滤波 sl_WlanRxFilterSet()。sl_WlanRxFilterSet()的调用语句为

```
lRetVal = sl_WlanRxFilterSet(SL_REMOVE_RX_FILTER, (_u8 *)&RxFilterIdMask,
                             sizeof(_WlanRxFilterOperationCommandBuff_t));
```

sl_WlanRxFilterSet()的函数格式为

```
_i16 sl_WlanRxFilterSet(const SLrxFilterOperation_t RxFilterOperation,
                        const _u8* const pInputBuffer,
                        _u16 InputbufferLength);
```

功能：设置接收滤波参数。

参数说明：

- RxFilterOperation：操作，传递值是 SL_REMOVE_RX_FILTER（删除接收滤波），在 wlan_rx_filters.h 中定义为 1。
- pInputBuffer：输入缓存指针，传递变量名是 RxFilterIdMask。
- InputbufferLength：输入缓存字节数，传递值是 20。

返回值：0 表示成功，负值表示错误。

sl_WlanRxFilterSet()的声明在 wlan_rx_filters.h 中，代码在 wlan.c 中。sl_WlanRxFilterSet()的返回值 lRetVal 是 0：设置接收滤波参数成功。

（12）停止 SimpleLink 函数 sl_Stop()。sl_Stop()的调用语句为

```
lRetVal = sl_Stop(SL_STOP_TIMEOUT);
```

sl_Stop()的函数格式为

```
_i16 sl_Stop(_u16 timeout);
```

功能：停止 SimpleLink。

参数说明：timeout，超时，传递值是 SL_STOP_TIMEOUT，在 main.h 中定义为 30。

返回值：0 表示成功，负值表示错误。

sl_Stop()的声明在 device.h 中，代码在 device.c 中。sl_Stop()的返回值 lRetVal 是 0：停止 SimpleLink 成功。

ConfigureSimpleLinkToDefaultState()的返回值 lRetVal 是 0，程序调用 UART_PRINT()在超级终端中显示下列信息。

```
Device is configured in default state      设备配置为默认状态
```

3．启动 SimpleLink sl_Start()

启动 SimpleLink 的函数是 sl_Start()，返回值 lRetVal 是 0，表示启动 SimpleLink 进入站点模式，程序调用 UART_PRINT()在超级终端中显示下列信息。

```
Device started as STATION                    设备作为站点启动
```

4．连接 WLAN WlanConnect()

连接 WLAN 的函数是 WlanConnect()，通过调用 sl_WlanConnect()实现，如图 5.5 所示。

```
754 static long WlanConnect()
755 {
756     SlSecParams_t secParams = {0};
757     long lRetVal = 0;
758
759     secParams.Key = (signed char*)SECURITY_KEY;
760     secParams.KeyLen = strlen(SECURITY_KEY);
761     secParams.Type = SECURITY_TYPE;
762
763     lRetVal = sl_WlanConnect((signed char*)SSID_NAME, strlen(SSID_NAME), 0, &secParams, 0);
764     ASSERT_ON_ERROR(lRetVal);
765
766     // Wait for WLAN Event
767     while((!IS_CONNECTED(g_ulStatus)) || (!IS_IP_ACQUIRED(g_ulStatus)))
768     {
769         // Toggle LEDs to Indicate Connection Progress
770         GPIO_IF_LedOff(MCU_IP_ALLOC_IND);
771         MAP_UtilsDelay(800000);
772         GPIO_IF_LedOn(MCU_IP_ALLOC_IND);
773         MAP_UtilsDelay(800000);
774     }
775
776     return SUCCESS;
777
778 }
```

图 5.5　Wlan_station 项目 WlanConnect()函数

sl_WlanConnect()的函数格式为

```
_i16 sl_WlanConnect(_i8* pName, _i16 NameLen, _u8 *pMacAddr,
                    SlSecParams_t* pSecParams,
                    SlSecParamsExt_t* pSecExtParams);
```

功能：连接 WLAN。

参数说明：

- pName：接入点名称指针，接入点名称 SSID_NAME 需要用户根据自己的实际情况在 common.h 中设置（如设置为“1234”）。
- NameLen：接入点名称长度，传递值是 strlen(SSID_NAME)。
- pMacAddr：MAC 地址指针，传递值是 0。
- pSecParams：安全参数指针，安全参数传递变量名是 secParams，安全参数包括安全类型、安全密码和密码长度，安全类型和安全密码需要用户根据自己的实际情况在 common.h 中设置。安全类型主要包括下列值。
 - ✧ SL_SEC_TYPE_OPEN：开放，0。
 - ✧ SL_SEC_TYPE_WEP：WEP 加密，1。
 - ✧ SL_SEC_TYPE_WPA_WPA2：WPA 加密，2。
- pSecExtParams：企业参数指针，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_WlanConnect()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanConnect()运行时调用 WLAN 事件处理函数 SimpleLinkWlanEventHandler()和 NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()处理 WLAN 事件和 NETAPP 事件。

为了分析 WLAN 事件处理函数 SimpleLinkWlanEventHandler()和 NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()，需在两个处理函数的第一条语句设置断点（参见图 5.6 和图 5.8），同时在 WlanConnect()函数的下列语句设置断点（参见图 5.5）。

```
while((!IS_CONNECTED(g_ulStatus)) || (!IS_IP_ACQUIRED(g_ulStatus)))
```

WLAN 事件处理函数 SimpleLinkWlanEventHandler()的部分内容如图 5.6 所示。

```
236 void SimpleLinkWlanEventHandler(SlWlanEvent_t *pWlanEvent)
237 {
238     switch(pWlanEvent->Event)
239     {
240         case SL_WLAN_CONNECT_EVENT:
241         {
242             SET_STATUS_BIT(g_ulStatus, STATUS_BIT_CONNECTION);
```

图 5.6　Wlan_station 项目 SimpleLinkWlanEventHandler()函数（部分）

```
void SimpleLinkWlanEventHandler(SlWlanEvent_t *pWlanEvent);
```

功能：WLAN 事件处理。

参数说明：pWlanEvent，WLAN 事件结构体指针，WLAN 事件结构体如图 5.7 所示。

Expressions

Expression	Type	Value
pWlanEvent	struct <unnamed> *	0x200213E0
*(pWlanEvent)	struct <unnamed>	{...}
Event	unsigned long	1
EventData	union <anonymous_union>	{...}
smartConfigStartResponse	struct <unnamed>	{...}
smartConfigStopResponse	struct <unnamed>	{...}
APModeStaConnected	struct <unnamed>	{...}
APModestaDisconnected	struct <unnamed>	{...}
STAandP2PModeWlanConnected	struct <unnamed>	{...}
connection_type	unsigned char	.
ssid_len	unsigned char	.
ssid_name	unsigned char[32]	0x200213E6
go_peer_device_name_len	unsigned char	.
go_peer_device_name	unsigned char[32]	0x20021407
bssid	unsigned char[6]	0x20021427
reason_code	unsigned char	.
padding	unsigned char[2]	0x2002142E
STAandP2PModeDisconnected	struct <unnamed>	{...}
P2PModeDevFound	struct <unnamed>	{...}
P2PModeNegReqReceived	struct <unnamed>	{...}
P2PModewlanConnectionFailure	struct <unnamed>	{...}

图 5.7　Wlan_station 项目 WLAN 事件结构体

注意：WLAN 事件结构体的显示有下列两种方法。

（1）将鼠标移到 SimpleLinkWlanEventHandler()函数中的“pWlanEvent”处，在弹出“Expression”（表达式）窗口中显示（参见图 3.29）。

（2）在 SimpleLinkWlanEventHandler()函数中选择“pWlanEvent”并右击，在弹出菜单中单击“Add Watch Expression…”（添加监视表达式），在添加监视表达式对话框中单击“OK”按钮，在“Expression”（表达式）标签中显示（参见图 5.7）。

WLAN 事件结构体中 Event（事件）主要包括下列值。

- SL_WLAN_CONNECT_EVENT：WLAN 连接事件，1。
- SL_WLAN_DISCONNECT_EVENT：WLAN 断开事件，2。

对于 WLAN 连接事件（Event=1），WLAN 事件数据 STAandP2PModeWlanConnected 主要包括 ssid_name（接入点名称）和 bssid（接入点 MAC 地址）。

WLAN 事件处理函数首先把设备状态 g_ulStatus 设置为 STATUS_BIT_ CONNECTION（连接），然后调用 UART_PRINT()在超级终端中显示下列信息。

```
[WLAN EVENT] STA Connected to the AP: 1234 ,BSSID: 3c:e5:a6:1:7:bc
[WLAN 事件] 站点连接到接入点：1234，BSSID: 3c:e5:a6:1:7:bc
```

NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()的部分内容如图 5.8 所示。

```
323 void SimpleLinkNetAppEventHandler(SlNetAppEvent_t *pNetAppEvent)
324 {
325     switch(pNetAppEvent->Event)
326     {
327         case SL_NETAPP_IPV4_IPACQUIRED_EVENT:
328         {
329             SlIpV4AcquiredAsync_t *pEventData = NULL;
330
331             SET_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_AQUIRED);
332
333             //Ip Acquired Event Data
334             pEventData = &pNetAppEvent->EventData.ipAcquiredV4;
335
336             //Gateway IP address
337             g_ulGatewayIP = pEventData->gateway;
```

图 5.8　Wlan_station 项目 SimpleLinkNetAppEventHandler()函数（部分）

```
void SimpleLinkNetAppEventHandler(SlNetAppEvent_t *pNetAppEvent);
```

功能：NETAPP 事件处理。

参数说明：pNetAppEvent，NETAPP 事件结构体指针，NETAPP 事件结构体如图 5.9 所示。

Expressions

Expression	Type	Value
pNetAppEvent	struct <unnamed> *	0x200213E0
*(pNetAppEvent)	struct <unnamed>	{...}
Event	unsigned long	1
EventData	union <anonymous_union>	{...}
ipAcquiredV4	struct <unnamed>	{...}
ip	unsigned long	0xC0A80106 (Hex)
gateway	unsigned long	0xC0A80101 (Hex)
dns	unsigned long	0xCA6A2E97 (Hex)
ipAcquiredV6	struct <unnamed>	{...}
sd	unsigned long	3232235782
ipLeased	struct <unnamed>	{...}
ipReleased	struct <unnamed>	{...}

图 5.9　Wlan_station 项目 NETAPP 事件结构体

其中 Event（事件）主要包括下列值。

- SL_NETAPP_IPV4_IPACQUIRED_EVENT：获得 IPV4 事件，1。
- SL_NETAPP_IPV6_IPACQUIRED_EVENT：获得 IPV6 事件，2。

对于获得 IPV4 事件（Event=1），NETAPP 事件数据 ipAcquiredV4 主要包括 ip（获取 IP）和 gateway（网关 IP 地址）。

NETAPP 事件处理函数首先把设备状态 g_ulStatus 设置为 STATUS_BIT_IP_AQUIRED（获得 IP），然后调用 UART_PRINT()在超级终端中显示下列信息。

```
[NETAPP EVENT] IP Acquired: IP=192.168.1.6 , Gateway=192.168.1.1
[NETAPP 事件] 获得 IP=192.168.1.6, 网关=192.168.1.1（十六进制 0xC0A80101）
```

sl_WlanConnect()运行后等待 WLAN 事件，根据设备状态 g_ulStatus 判断是否连接成功 IS_CONNECTED(g_ulStatus)或获得 IP 成功 IS_IP_ACQUIRED(g_ulStatus)，如果没有成功则等待，开发板上的红 LED 闪烁，直到连接成功，此时红 LED 点亮，退出 WlanConnect()，返回值 lRetVal 为 0，表示 WLAN 连接成功，程序调用 UART_PRINT()显示下列信息。

```
Connection established w/ AP and IP is aquired    建立与 AP 的连接并获得 IP
Pinging...!
```

5．检查 LAN 连接 CheckLanConnection()

检查 LAN 连接的函数是 CheckLanConnection()，通过调用 sl_NetAppPingStart()实现对接入点的 Ping 操作，sl_NetAppPingStart()调用语句为

```
lRetVal = sl_NetAppPingStart((SlPingStartCommand_t*)&pingParams,
   SL_AF_INET, (SlPingReport_t*)&pingReport, SimpleLinkPingReport);
```

sl_NetAppPingStart()函数格式为

```
_i16 sl_NetAppPingStart(SlPingStartCommand_t* pPingParams, _u8 family,
   SlPingReport_t *pReport, const P_SL_DEV_PING_CALLBACK pPingCallback);
```

功能：启动 Ping。

参数说明：

- pPingParams：Ping 参数指针，传递变量名是 pingParams。
- family：Ping 地址类型，传递值是 SL_AF_INET，在 socket.h 中定义为 2。
- pReport：Ping 报告指针，传递变量名是 pingReport。
- pPingCallback：回调函数指针，传递值是 SimpleLinkPingReport。

返回值：0 表示成功，负值表示错误。

sl_NetAppPingStart()的声明在 netapp.h 中，代码在 netapp.c 中。当 sl_NetAppPingStart()运行后，回调函数 SimpleLinkPingReport()将设备状态 g_ulStatus 设置为 STATUS_BIT_PING_DONE（已 Ping），并将报告中的接收数据包数赋值给 g_ulPing PacketsRecv（值为 3）。

CheckLanConnection()检查设备状态 g_ulStatus 和接收数据包数 g_ulPingPacketsRecv，连接成功时返回 0，表示检查 LAN 连接成功，开发板上的绿 LED 点亮。

6．检查 Internet 连接 CheckInternetConnection()

检查 Internet 连接的函数是 CheckInternetConnection()，首先调用 sl_NetAppDnsGetHostByName()获取主机 IP，然后调用 sl_NetAppPingStart()实现对主机的 Ping 操作。

获取主机 IP 的调用语句为

```
lRetVal = sl_NetAppDnsGetHostByName((signed char*)HOST_NAME,
                                    sizeof(HOST_NAME), &ulIpAddr, SL_AF_INET);
```

sl_NetAppDnsGetHostByName()函数格式为

```
_i16 sl_NetAppDnsGetHostByName(_i8 * hostname, _u16 usNameLen,
                               _u32* out_ip_addr, _u8 family);
```

功能：获取主机 IP。

参数说明：

- hostname：主机名，传递变量名是 HOST_NAME，传递值是“www.ti.com”。
- usNameLen：主机名长度，传递值是 sizeof(HOST_NAME)。
- out_ip_addr：IP 地址指针，存放获取的 IP 地址，传递变量名是 ulIpAddr。
- family：地址类型，传递值是 SL_AF_INET，在 socket.h 中定义为 2。

返回值：0 表示成功，负值表示错误。

sl_NetAppDnsGetHostByName()的声明在 netapp.h 中，代码在 netapp.c 中。检查 Internet 连接成功后，开发板上的黄 LED 点亮。程序调用 UART_PRINT()在超级终端中显示下列信息。

```
Device pinged both the gateway and the external host
设备 Ping 通网关和外部主机
WLAN STATION example executed successfully
WLAN STATION 例程成功运行
```

7．停止 SimpleLink sl_Stop()

停止 SimpleLink 的函数是 sl_Stop()，返回值 lRetVal 是 0，表示停止 SimpleLink 成功，程序进入死循环。

综上所述，Wlan_station 项目使用的 SimpleLink API 函数如表 5.1 所示。

表 5.1　Wlan_station 项目使用的 SimpleLink API 函数

功　能	函数名	调用函数	代码文件
启动 SimpleLink	**sl_Start()**	ConfigureSimpleLinkToDefaultState() WlanStationMode()	device.c
获取设备配置	sl_DevGet()	ConfigureSimpleLinkToDefaultState()	device.c
设置 WLAN 策略	sl_WlanPolicySet()	ConfigureSimpleLinkToDefaultState()	wlan.c
删除配置文件	sl_WlanProfileDel()	ConfigureSimpleLinkToDefaultState()	wlan.c
断开 WLAN 连接	sl_WlanDisconnect()	ConfigureSimpleLinkToDefaultState()	wlan.c
设置网络配置	sl_NetCfgSet()	ConfigureSimpleLinkToDefaultState()	netcfg.c
设置 WLAN 配置	sl_WlanSet()	ConfigureSimpleLinkToDefaultState()	wlan.c
停止 SimpleLink	**sl_Stop()**	ConfigureSimpleLinkToDefaultState() WlanStationMode()	device.c
连接 WLAN	**sl_WlanConnect()**	WlanConnect()	wlan.c
启动 Ping	sl_NetAppPingStart()	CheckLanConnection()、CheckInternetConnection()	netapp.c
获取主机 IP	sl_NetAppDnsGetHostByName()	CheckInternetConnection()	netapp.c

Wlan_station 项目使用的 SimpleLink API 函数中，最重要的是 sl_Start()、sl_Stop()和 sl_WlanConnect()，声明如下。

```
_i16 sl_Start(const void* pIfHdl, _i8* pDevName,
              const P_INIT_CALLBACK pInitCallBack);
```

```
_i16 sl_Stop(_u16 timeout);
_i16 sl_WlanConnect(_i8* pName, _i16 NameLen, _u8 *pMacAddr,
                    SlSecParams_t* pSecParams,
                    SlSecParamsExt_t* pSecExtParams);
```

功能：连接 WLAN。

参数说明：

- pName：接入点名称指针。
- NameLen：接入点名称长度，传递值是 strlen(SSID_NAME)。
- pMacAddr：MAC 地址指针，传递值是 0。
- pSecParams：安全参数指针，安全参数包括安全类型、安全密码和密码长度，安全类型主要包括下列值：
 - ✧ SL_SEC_TYPE_OPEN：开放，0。
 - ✧ SL_SEC_TYPE_WEP：WEP 加密，1。
 - ✧ SL_SEC_TYPE_WPA_WPA2：WPA 加密，2。
- pSecExtParams：企业参数指针，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_WlanConnect()的声明在 wlan.h 中，代码在 wlan.c 中。sl_WlanConnect()运行时调用 WLAN 事件处理函数 SimpleLinkWlanEventHandler()和 NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()处理 WLAN 事件和 NETAPP 事件。

WLAN 连接时的 WLAN 事件主要包括下列值。

- SL_WLAN_CONNECT_EVENT：WLAN 连接事件，1。
- SL_WLAN_DISCONNECT_EVENT：WLAN 断开事件，2。

对于 WLAN 连接事件，WLAN 事件数据 STAandP2PModeWlanConnected 主要包括 ssid_name（接入点名称）和 bssid（接入点 MAC 地址）。

WLAN 事件处理函数首先把设备状态 g_ulStatus 设置为 STATUS_BIT_CONNECTION（连接），然后调用 UART_PRINT()在超级终端中显示下列信息。

```
[WLAN EVENT] STA Connected to the AP: 1234 ,BSSID: 3c:e5:a6:1:7:bc
[WLAN 事件] 站点连接到接入点：1234，BSSID: 3c:e5:a6:1:7:bc
```

WLAN 连接时的 NETAPP 事件主要包括下列值。

- SL_NETAPP_IPV4_IPACQUIRED_EVENT：获得 IPv4 事件，1。
- SL_NETAPP_IPV6_IPACQUIRED_EVENT：获得 IPv6 事件，2。

对于获得 IPv4 事件，NETAPP 事件数据 ipAcquiredV4 主要包括 ip（获得 IP）和 gateway（网关 IP 地址）。

NETAPP 处理函数首先把设备状态 g_ulStatus 设置为 STATUS_BIT_IP_ AQUIRED（获得 IP），然后调用 UART_PRINT()在超级终端中显示下列信息。

```
[NETAPP EVENT] IP Acquired: IP=192.168.1.6 , Gateway=192.168.1.1
[NETAPP 事件] 获得 IP=192.168.1.6, 网关=192.168.1.1
```

5.2　Wlan_ap 示例程序分析

Wlan_ap 示例程序在“example\getting_started_with_wlan_ap”目录中，Wlan_ap 项目的工作示意图如图 5.10 所示。



图 5.10　Wlan_ap 项目工作示意图

Wi-Fi 站点首先连接 CC3200 接入点，然后 CC3200 接入点 Ping Wi-Fi 站点。

Wlan_ap 项目包含下列 3 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。

Wlan_ap 项目程序流程图如图 5.11 所示。

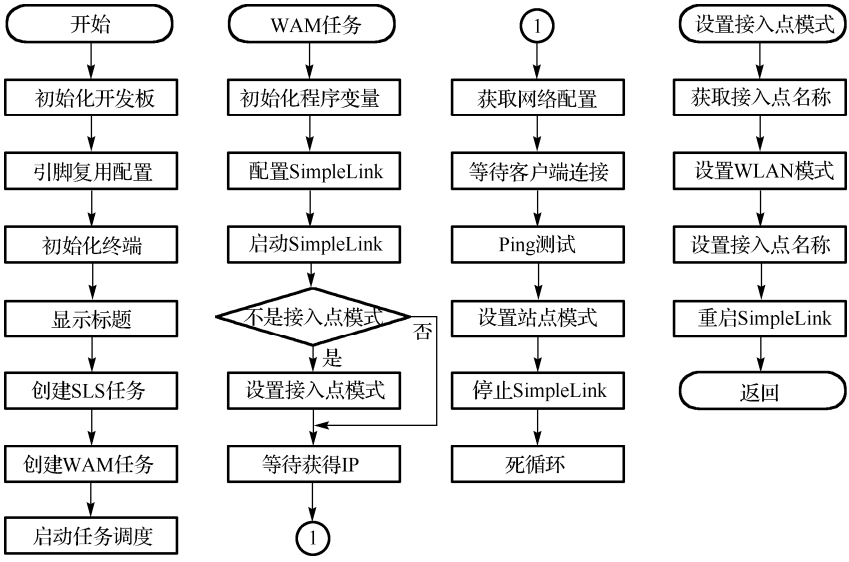


图 5.11　Wlan_ap 项目程序流程图

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（2）在 PC 上打开超级终端 Tera Term。

（3）在 CCS Project Explorer（项目浏览器）中右击 wlan_ap 项目，在弹出菜单中选择“Build Project”生成目标文件 wlan_ap.out。

（4）在 CCS Project Explorer（项目浏览器）中右击 wlan_ap 项目，在弹出菜单中选择“Debug As”→“Code Composer Debug Session”，将 wlan_ap.out 下载到开发板的 CC3200 中，然后进入调试界面。

（5）单击调试工具栏中的“Resume”（运行）按钮运行程序，超级终端中显示下列信息。

```
*********************************************************************
                    CC3200 WLAN AP Application
*********************************************************************

Host Driver Version: 1.0.0.10
Build Version 2.4.0.2.31.1.3.0.1.1.0.3.34
Device is configured in default state
Device started as STATION
Enter the AP SSID name:                          输入接入点名称
```

（6）在超级终端中输入接入点名称（如“asdf”），按 Enter（回车）键后继续显示

```
Device is configured in AP mode                  设备配置为接入点模式
Connect a client to Device                       连接客户端到设备
```

（7）在手机的设定中打开 Wi-Fi，找到名称为“asdf”的接入点连接，超级终端显示：

```
[NETAPP EVENT] IP Leased to Client: IP=192.168.1.2
[NETAPP 事件] 客户端分配 IP=192.168.1.2
Client is connected to Device                    客户端已连接到设备
Running Ping Test...                             运行 Ping 测试
Ping Test successful                             Ping 测试成功
WLAN AP example executed successfully            WLAN AP 例程成功运行
```

（8）单击调试工具栏中的“Terminate”（结束）按钮，结束调试返回编辑界面。

（9）重复（4）重新进入调试界面，为程序的分析做准备。

下面详细介绍程序的分析步骤，包括主函数 main()分析和 WLAN 接入点模式（WAM）函数 WlanAPMode()分析。

5.2.1 主函数 main()分析

主函数 main()包括初始化开发板、引脚复用配置、初始化终端、显示标题、创建 SLS（SimpleLinkSpawn）任务 VStartSimpleLinkSpawnTask()、创建 WAM（WlanAPMode）任务 osi_TaskCreate()和启动任务调度 osi_start()等。

Wlan_ap 项目主函数 main()的分析方法和 wlan_station 项目基本相同，不同之处主要有以下两点。

（1）调用 Report()在超级终端中显示下列标题。

```
*********************************************************************
                    CC3200 WLAN AP Application
*********************************************************************
```

（2）调用 osi_TaskCreate()创建 WAM（WlanAPMode）任务，调用语句为

```
lRetVal = osi_TaskCreate(WlanAPMode,
                         (const signed char*)"wireless LAN in AP mode",
                          OSI_STACK_SIZE, NULL, 1, NULL);
```

为了实现 WAM 任务 WlanAPMode 的分析，在运行 osi_start()前，需先在 WlanAPMode()的下列语句设置断点。

```
InitializeAppVariables();
```

5.2.2　WlanAPMode()分析

WlanAPMode()是 wlan_ap 项目的核心函数，主要包括下列内容，其中最主要的是 sl_Start()、sl_WlanSetMode()和 sl_Stop()。

- 初始化程序变量：InitializeAppVariables()。
- 配置 SimpleLink 为默认状态：ConfigureSimpleLinkToDefaultState()。
- 启动 SimpleLink：sl_Start()。
- 设置接入点模式：ConfigureMode()（sl_WlanSetMode()）。
- 等待获得 IP：STATUS_BIT_IP_AQUIRED。
- 获取网络配置（可以省略）：sl_NetCfgGet()。
- 等待客户端连接：STATUS_BIT_IP_ LEASED。
- Ping 测试：sl_NetAppPingStart()。
- 设置站点模式：sl_WlanSetMode()。
- 停止 SimpleLink：sl_Stop()。

1．初始化程序变量 InitializeAppVariables()

初始化程序变量的函数是 InitializeAppVariables()，函数格式为

```
static void InitializeAppVariables();
```

功能：初始化程序变量。

初始化程序变量主要对下列全局变量进行初始化。

- g_ulStatus：设备状态，0。
- g_ulStaIP：站点 IP 地址，0。
- g_ulPingPacketsRecv：接收数据包数，0。
- g_ulGatewayIP：网关 IP 地址，0。

2．配置 SimpleLink 为默认状态 ConfigureSimpleLinkToDefaultState()

配置 SimpleLink 为默认状态的函数是 ConfigureSimpleLinkToDefaultState()，Wlan_ap 项目 ConfigureSimpleLinkToDefaultState()的内容和 wlan_station 项目相同，这里不再赘述。

3．启动 SimpleLink 函数 sl_Start()

启动 SimpleLink 的函数是 sl_Start()。

4．设置接入点模式 ConfigureMode()

由于设备不是接入点模式（ROLE_AP），需要调用 ConfigureMode()设置接入点模式，如图 5.12 所示。

```
818    //
819    // Configure the networking mode and ssid name(for AP mode)
820    //
821    if(lRetVal != ROLE_AP)
822    {
823        if(ConfigureMode(lRetVal) != ROLE_AP)
824        {
825            UART_PRINT("Unable to set AP mode, exiting Application...\n\r");
826            sl_Stop(SL_STOP_TIMEOUT);
827            LOOP_FOREVER();
828        }
829    }
830
831    while(!IS_IP_ACQUIRED(g_ulStatus))
832    {
833      //looping till ip is acquired
834    }
```

图 5.12　Wlan_ap 项目 WlanAPMode()函数（部分）

设置接入点模式 ConfigureMode()主要包括下列内容，其中最主要的是 sl_WlanSetMode()。

- 获取接入点名称：GetSsidName()。
- 设置 WLAN 模式：sl_WlanSetMode()。
- 设置接入点名称：sl_WlanSet()。
- 重启 SimpleLink：sl_Stop() + sl_Start()。

（1）获取接入点名称 GetSsidName()。设置接入点模式 ConfigureMode()首先在超级终端中显示下列信息。

```
Enter the AP SSID name:                          输入接入点名称
```

然后调用获取接入点名称 GetSsidName()从超级终端获取接入点名称，当用户在超级终端中输入接入点名称（如“asdf”）并按 Enter（回车）键后，GetSsidName()将接入点名称存入数组 pcSsidName[]中。

（2）设置 WLAN 模式 sl_WlanSetMode()。sl_WlanSetMode()的调用语句为

```
lRetVal = sl_WlanSetMode(ROLE_AP);
```

sl_WlanSetMode()的函数格式为

```
_i16 sl_WlanSetMode(const _u8 char mode);
```

功能：设置 WLAN 模式。

参数说明：mode，WLAN 模式，包含下列值。

- WLAN_SET_STA_MODE：站点模式，1。
- WLAN_SET_AP_MODE：接入点模式，2。
- WLAN_SET_P2P_MODE：点对点模式，3。

传递值是 ROLE_AP，在 wlan.h 中定义为 2，表示接入点模式。

返回值：0 表示设置成功。

sl_WlanSetMode()的声明在 wlan.h 中，代码在 wlan.c 中。

（3）设置接入点名称 sl_WlanSet()。sl_WlanSet()的调用语句为

```
lRetVal = sl_WlanSet(SL_WLAN_CFG_AP_ID, WLAN_AP_OPT_SSID,
                     strlen(pcSsidName), (unsigned char*)pcSsidName);
```

sl_WlanSet()的函数格式为

```
_i16 sl_WlanSet(_u16 ConfigId, _u16 ConfigOpt, _u16 ConfigLen, _u8 *pValues);
```

功能：设置 WLAN 配置。

参数说明：

- ConfigId：配置 ID，传递值是 SL_WLAN_CFG_AP_ID（接入点 ID），在 wlan.h 中定义为 0。
- ConfigOpt：配置选项，传递值是 WLAN_AP_OPT_SSID（接入点名称），在 wlan.h 中定义为 0。
- ConfigLen：配置长度，传递值是接入点名称长度 strlen(pcSsidName)。
- pValues：配置值指针，传递变量名是 pcSsidNamer，传递值是用户在超级终端中输入的接入点名称（如“asdf”）。

返回值：0 表示成功，负值表示错误。

sl_WlanSet()的声明在 wlan.h 中，代码在 wlan.c 中。设置接入点名称后，程序调用 UART_PRINT()在超级终端中显示下列信息。

```
Device is configured in AP mode                    设备配置为接入点模式
```

（4）重启 SimpleLink。设备配置为接入点模式后，必须重启 SimpleLink 才能生效。重启 SimpleLink 通过调用停止 SimpleLink 函数 sl_Stop()和启动 SimpleLink 函数 sl_Start()实现。重启 SimpleLink 配置为接入点模式时，系统调用 NETAPP 事件处理函数 SimpleLink NetAppEventHandler()处理 NETAPP 事件。

为了分析 WLAN 事件处理函数 SimpleLinkWlanEventHandler()和 NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()，需在两个处理函数的第一条语句设置断点，同时在 WlanAPMode()函数的下列语句设置断点（参见图 5.12）。

```
while(!IS_IP_ACQUIRED(g_ulStatus))
```

NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()的部分内容如图 5.13 所示。

```
325 void SimpleLinkNetAppEventHandler(SlNetAppEvent_t *pNetAppEvent)
326 {
327     switch(pNetAppEvent->Event)
328     {
329         case SL_NETAPP_IPV4_IPACQUIRED_EVENT:
330         case SL_NETAPP_IPV6_IPACQUIRED_EVENT:
331         {
332             SET_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_AQUIRED);
333         }
334         break;
```

图 5.13　Wlan_ap 项目 SimpleLinkNetAppEventHandler()函数（部分）

配置为接入点模式时的 NETAPP 事件主要包括下列值。

- SL_NETAPP_IPV4_IPACQUIRED_EVENT：获得 IPV4 事件，1。
- SL_NETAPP_IPV6_IPACQUIRED_EVENT：获得 IPV6 事件，2。

对于获得 IPv4 事件或获得 IPv6 事件，处理函数把设备状态 g_ulStatus 设置为 STATUS_BIT_IP_AQUIRED（获得 IP）。

5．等待获得 IP

等待获得 IP 的语句为

```
while(!IS_IP_ACQUIRED(g_ulStatus))
```

如果没有获得 IP，程序循环等待，直到获得 IP 为止。由于 NETAPP 事件处理函数已将 g_ulStatus 设置为 STATUS_ BIT_IP_AQUIRED（获得 IP），所以程序并不循环等待。

6．获取网络配置 sl_NetCfgGet()

sl_NetCfgGet()的调用语句为

```
lRetVal = sl_NetCfgGet(SL_IPV4_STA_P2P_CL_GET_INFO, &ucDHCP, &len,
                       (unsigned char *)&ipV4);
```

sl_NetCfgGet()的函数格式为

```
long sl_NetCfgGet(unsigned char ConfigId, unsigned char *pConfigOpt,
                  unsigned char *pConfigLen, unsigned char *pValues);
```

功能：获取网络配置。

参数说明：

- ConfigId：配置 ID，传递值是 SL_IPV4_STA_P2P_CL_GET_INFO（获取信息），在 netcfg.h 中定义为 3。
- ConfigOpt：配置选项，传递变量名是 ucDHCP，传递值是 1。
- ConfigLen：配置长度，传递变量名是 len，传递值是 0x10。
- pValues：配置值指针，传递变量名是 ipV4，用于获取 IP 地址。

返回值：0 表示成功，负值表示错误。

sl_NetCfgGet()的声明在 netcfg.h 中，代码在 netcfg.c 中。

7．等待客户端连接

（1）在 WlanAPMode()中的下列语句处设置断点。

```
iTestResult = PingTest(g_ulStaIp);
```

（2）单击调试工具栏中的“Resume”（运行）按钮，运行程序，获取网络配置后，程序调用 UART_PRINT()在超级终端中显示下列信息。

```
Connect a client to Device                    连接客户端到设备
```

在手机的设定中打开 Wi-Fi，找到名称为“asdf”的接入点连接。

连接时程序调用 WLAN 事件处理函数 SimpleLinkWlanEventHandler()和 NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()处理 WLAN 事件和 NETAPP 事件。

WLAN 事件处理函数 SimpleLinkWlanEventHandler()的部分内容如图 5.14 所示。

```
273         case SL_WLAN_STA_CONNECTED_EVENT:
274         {
275             // when device is in AP mode and any client connects to device cc3xxx
276             SET_STATUS_BIT(g_ulStatus, STATUS_BIT_CONNECTION);
```

图 5.14　Wlan_ap 项目 SimpleLinkWlanEventHandler()函数（部分）

客户端连接接入点时的 WLAN 事件主要包括下列值。

对于 WLAN 站点已连接事件，处理函数把设备状态 g_ulStatus 设置为 STATUS_BIT_CONNECTION（连接）。

（3）单击调试工具栏中的“Resume”（运行）按钮，运行程序。

NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()的部分内容如图 5.15 所示。

```
325 void SimpleLinkNetAppEventHandler(SlNetAppEvent_t *pNetAppEvent)
326 {
327     switch(pNetAppEvent->Event)
328     {
329         case SL_NETAPP_IPV4_IPACQUIRED_EVENT:
330         case SL_NETAPP_IPV6_IPACQUIRED_EVENT:
331         {
332             SET_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_AQUIRED);
333         }
334         break;
335
336         case SL_NETAPP_IP_LEASED_EVENT:
337         {
338             SET_STATUS_BIT(g_ulStatus, STATUS_BIT_IP_LEASED);
339
340             g_ulStaIp = (pNetAppEvent)->EventData.ipLeased.ip_address;
```

图 5.15　Wlan_ap 项目 SimpleLinkNetAppEventHandler()函数（部分）

客户端连接接入点时的 NETAPP 事件主要包括下列值。

- SL_NETAPP_IP_LEASED_EVENT：分配 IP 事件，3。
- SL_WLAN_STA_CONNECTED_EVENT：WLAN 站点已连接事件，5。
- SL_WLAN_STA_DISCONNECTED_EVENT：WLAN 站点已断开事件，6。
- SL_NETAPP_IP_RELEASED_EVENT：释放 IP 事件，4。

对于分配 IP 事件，NETAPP 事件数据 ipLeased 主要包括 ip_address（IP 地址）、lease_time（分配时间）和 mac（MAC 地址）。

处理函数首先把设备状态 g_ulStatus 设置为 STATUS_BIT_IP_ LEASED（分配 IP），把站点 IP 赋值给 g_ulStaIP，然后调用 UART_PRINT()显示下列信息。

```
[NETAPP EVENT] IP Leased to Client: IP=192.168.1.2
[NETAPP 事件] 客户端分配 IP=192.168.1.2
```

（4）单击调试工具栏中的“Resume”（运行）按钮，运行程序。

WLAN 和 NETAPP 事件处理后，由于 NETAPP 事件处理函数已将 g_ulStatus 设置为 STATUS_BIT_IP_LEASED（分配 IP），程序退出下列循环。

```
while(!IS_IP_LEASED(g_ulStatus))
```

并调用 UART_PRINT()在超级终端中显示下列信息。

```
Client is connected to Device                客户端已连接到设备
```

8．Ping 测试 sl_NetAppPingStart()

Ping 测试的函数是 PingTest()，单击调试工具栏中的“Step Over”（单步跨越）按钮，运行 PingTest()，程序首先调用 UART_PRINT()在超级终端中显示下列信息。

```
Running Ping Test...                         运行 Ping 测试
```

然后调用 sl_NetAppPingStart()实现对客户端的 Ping 操作，具体操作和 wlan_station 项目相同，这里不再赘述。最后程序检查接收数据包数 g_ulPingPacketsRecv，连接成功后调用 UART_PRINT()在超级终端中显示下列信息。

```
Ping Test successful                         Ping 测试成功
```

9．设置站点模式并停止 SimpleLink

单击调试工具栏中的“Resume”（运行）按钮，运行设置站点模式函数 sl_Wlan SetMode()和停止 SimpleLink 函数 sl_Stop()，最后调用 UART_PRINT()显示下列信息并进入死循环。

```
WLAN AP example executed successfully        WLAN AP 例程成功运行
```

综上所述，Wlan_ap 项目使用的 SimpleLink API 函数如表 5.2 所示。

表 5.2　Wlan_ap 项目使用的 SimpleLink API 函数

功　能	函数名	调用函数	代码文件
启动 SimpleLink	**sl_Start()**	ConfigureSimpleLinkToDefaultState()、WlanAPMode()、ConfigureMode()	device.c
获取设备信息	sl_DevGet()	ConfigureSimpleLinkToDefaultState()	device.c
设置 WLAN 策略	sl_WlanPolicySet()	ConfigureSimpleLinkToDefaultState()	wlan.c
删除配置文件	sl_WlanProfileDel()	ConfigureSimpleLinkToDefaultState()	wlan.c
设置 WLAN 模式	**sl_WlanSetMode()**	ConfigureSimpleLinkToDefaultState()、ConfigureMode()、WlanAPMode()	wlan.c
断开 WLAN 连接	sl_WlanDisconnect()	ConfigureSimpleLinkToDefaultState()	wlan.c
设置网络配置	sl_NetCfgSet()	ConfigureSimpleLinkToDefaultState()	netcfg.c
设置 WLAN 配置	sl_WlanSet()	ConfigureSimpleLinkToDefaultState()、ConfigureMode()	wlan.c
停止 SimpleLink	**sl_Stop()**	ConfigureSimpleLinkToDefaultState()、ConfigureMode()、WlanAPMode()、	device.c
获取网络配置	sl_NetCfgGet()	WlanAPMode()	netcfg.c
启动 Ping	sl_NetAppPingStart()	PingTest()	netapp.c

Wlan_ap 项目使用的 SimpleLink API 函数中，最重要的是 sl_Start()、sl_Stop()和 sl_WlanSetMode()，声明如下。

```
int sl_Start(const void* pIfHdl, char* pDevName,
             const P_INIT_CALLBACK pInitCallBack);
int sl_Stop(unsigned short timeout);
int sl_WlanSetMode(const unsigned char mode);
```

功能：设置 WLAN 模式。

参数说明：mode，WLAN 模式，包含下列值。

- WLAN_SET_STA_MODE：站点模式，1。
- WLAN_SET_AP_MODE：接入点模式，2。
- WLAN_SET_P2P_MODE：点对点模式，3。

返回值：0，表示设置成功。

设置 WLAN 为接入点模式时，系统调用 NETAPP 事件处理函数 SimpleLinkNetApp EventHandler()处理 NETAPP 事件。配置为接入点模式时的 NETAPP 事件主要包括下列值。

- SL_NETAPP_IPV4_IPACQUIRED_EVENT：获得 IPV4 事件，1。
- SL_NETAPP_IPV6_IPACQUIRED_EVENT：获得 IPV6 事件，2。

对于获得 IPv4 事件或获得 IPv6 事件，NETAPP 处理函数把设备状态 g_ulStatus 设置为 STATUS_BIT_IP_AQUIRED（获得 IP）。

客户端连接接入点时程序调用 WLAN 事件处理函数 SimpleLinkWlanEventHandler()和 NETAPP 事件处理函数 SimpleLinkNetAppEventHandler()处理 WLAN 事件和 NETAPP 事件。客户端连接接入点时的 WLAN 事件主要包括下列值。

- SL_WLAN_STA_CONNECTED_EVENT：WLAN 站点已连接事件，5。
- SL_WLAN_STA_DISCONNECTED_EVENT：WLAN 站点已断开事件，6。

对于 WLAN 站点已连接事件，WLAN 处理函数把设备状态 g_ulStatus 设置为 STATUS_BIT_CONNECTION（连接）。

客户端连接接入点时的 NETAPP 事件主要包括下列值。

- SL_NETAPP_IP_LEASED_EVENT：分配 IP 事件，3。
- SL_NETAPP_IP_RELEASED_EVENT：释放 IP 事件，4。

对于分配 IP 事件，NETAPP 事件数据 ipLeased 主要包括 ip_address（IP 地址）、lease_ time（分配时间）和 mac（MAC 地址）。

NETAPP 处理函数首先把设备状态 g_ulStatus 设置为 STATUS_BIT_IP_ LEASED（分配 IP），把分配 IP 赋值给 g_ulStaIP，然后调用 UART_PRINT()显示下列信息。

```
[NETAPP EVENT] IP Leased to Client: IP=192.168.1.2
[NETAPP 事件] 客户端分配 IP=192.168.1.2
```

第 6 章

网络应用基础

网络应用的基础操作是套接字（Socket），包括 TCP 套接字和 UDP 套接字，套接字的流程如图 6.1 所示。

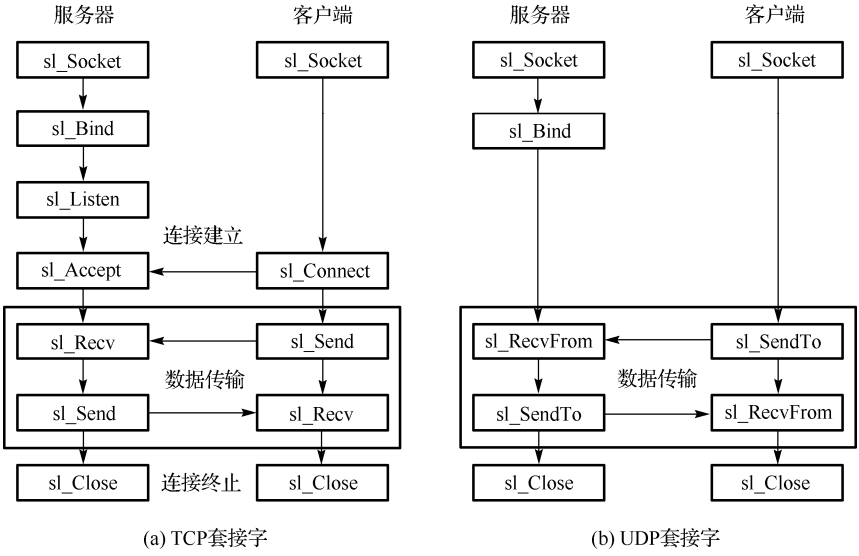


图 6.1　套接字流程

对于 TCP，服务器首先创建 TCP 套接字 sl_Socket，并将其绑定到 TCP 服务器地址 sl_Bind，然后在指定端口进行监听 sl_Listen，并接收 TCP 连接 sl_Accept，一旦建立 TCP 连接，就可以进行 TCP 数据接收 sl_Recv 和发送 sl_Send，最后关闭 TCP 套接字 sl_Close。

客户端首先创建 TCP 套接字 sl_Socket，并与服务器建立 TCP 连接 sl_Connect，一旦建立 TCP 连接，就可以进行 TCP 数据接收 sl_Recv 和发送 sl_Send，最后关闭 TCP 套接字 sl_Close。

对于 UDP，服务器只需创建 UDP 套接字 sl_Socket，并将其绑定到 UDP 服务器地址 sl_Bind，客户端只需创建 UDP 套接字 sl_Socket，即可以进行 UDP 数据接收 sl_RecvFrom 和发送 sl_SendTo，最后关闭 UDP 套接字 sl_Close。因此，相对 TCP 套接字，UDP 套接字简单，但是数据传输不可靠。要实现可靠的数据传输，必须使用 TCP 套接字。

套接字应用程序可以使用操作系统运行，也可以不使用操作系统运行。不使用操作系统运行时，通常通过调用 sl_SetSockOpt()设置为非阻塞模式；而使用操作系统运行时，由于多线程的实现，可以使用阻塞模式。

sl_Connect()、sl_Accept()、sl_Send()、sl_SendTo()、sl_Recv()和 sl_RecvFrom()的默认状态是阻塞模式，此时函数阻塞直到操作完成。非阻塞模式下，函数返回错误代码，此时对于必须完成的操作（如 sl_Connect()）通常使用循环实现阻塞。

本章首先分析两个基础示例程序：Tcp_socket 示例程序和 udp_socket 示例程序，然后在 wlan_ap 示例程序和 wlan_station 示例程序的基础上，结合 tcp_socket 示例程序，进行无线 UART 程序设计。

6.1 Tcp_socket 示例程序分析

Tcp_socket 示例程序在“example\tcp_socket”目录中，Tcp_socket 项目的工作示意图如图 6.2 所示。



图 6.2 Tcp_socket 项目工作示意图

CC3200 站点首先连接接入点，然后通过接入点与连接到接入点的 Wi-Fi 站点建立 TCP 套接字连接，最后通过 TCP 套接字实现 TCP 数据包的发送和接收。

Tcp_socket 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- udma_if.c：DMA 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Tcp_socket 项目程序流程如图 6.3 所示。

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）在安卓手机（或笔记本电脑）上打开 Wi-Fi 并连接到用户接入点（如 1234），查看接入点分配给手机的 IP 地址（如 192.168.1.9）。

（2）在安卓手机上安装并运行 iperf.apk。

（3）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（4）在 PC 上打开超级终端。

（5）将 main.c 中的下列宏定义

```
#define IP_ADDR         0xc0a8006E          /* 192.168.0.110 */
```

修改为接入点分配给手机的 IP 地址，例如

```
#define IP_ADDR         0xc0a80109          /* 192.168.1.9 */
```

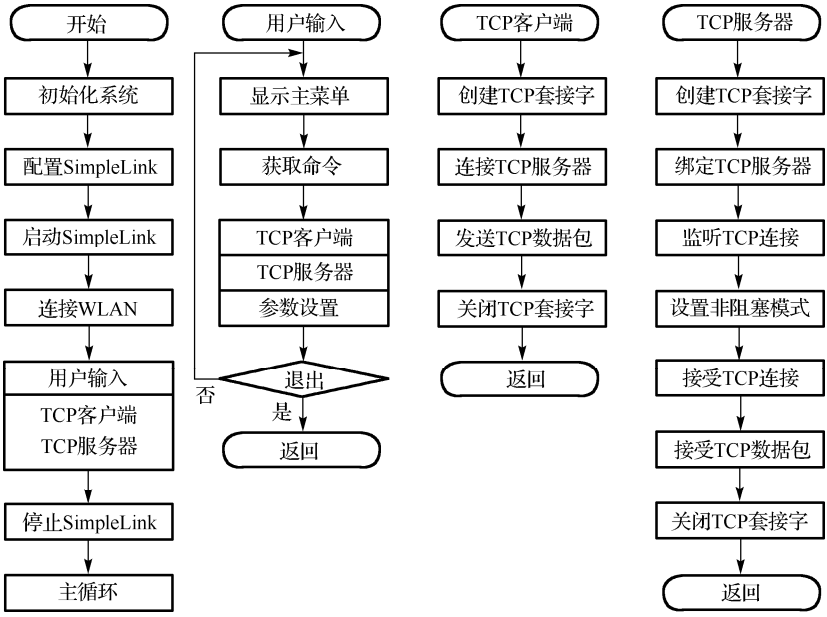


图 6.3　Tcp_socket 项目程序流程

（6）将 common.h 中的下列宏定义

```
#define SSID_NAME       "cc3200demo"        /* 接入点名称 */
#define SECURITY_TYPE   SL_SEC_TYPE_OPEN    /* 安全类型(OPEN/WEP/WPA) */
#define SECURITY_KEY    ""                  /* 安全密码 */
```

修改为自己 AP 的配置值，例如

```
#define SSID_NAME       "1234"              /* 接入点名称 */
#define SECURITY_TYPE   SL_SEC_TYPE_WPA     /* 安全类型(OPEN/WEP/WPA) */
#define SECURITY_KEY    "1234asdf"          /* 安全密码 */
```

注意：如果上述宏定义在 WLAN 站点示例程序分析中已经修改，此步可以省略。

（7）选中 tcp_socket 项目，按“F11”键，CCS 生成输出文件 tcp_socket.out，并将 tcp_socket.out 下载到开发板的 CC3200 中，然后进入调试界面。

（8）单击调试工具栏中的“Resume”（运行）按钮，运行程序，超级终端中显示下列信息。

```
*********************************************************************
                CC3200 TCP Socket Application
*********************************************************************

Host Driver Version: 1.0.0.10
Build Version 2.4.0.2.31.1.3.0.1.1.0.3.34
Device is configured in default state
Device started as STATION
Connecting to AP: 1234 ...                  连接接入点：1234
[WLAN EVENT] STA Connected to the AP: 1234 , BSSID: 3c:e5:a6:1:7:bc
[NETAPP EVENT] IP Acquired: IP=192.168.1.6 , Gateway=192.168.1.1
```

```
Connected to AP: 1234                          连接到接入点：1234
Device IP: 192.168.1.6

Default settings: SSID Name: 1234, PORT = 5001, Packet Count = 1000,
   Destination IP: 192.168.1.9

Choose Options:                                选择选项
1. Send TCP packets.                           1. 发送 TCP 数据包
2. Receive TCP packets.                        2. 接收 TCP 数据包
3. Settings.                                   3. 设置
4. Exit                                        4. 退出
Enter the option to use:                       输入使用的选项
```

（9）在超级终端中输入“3”，然后按“Enter”（回车）键选择“设置”，超级终端中显示“设置选项”子菜单。

```
Setting Options:                               设置选项
1. PORT                                        1. 端口
2. Packet Count                                2. 数据包数
3. Destination IP                              3. 目标 IP
4. Main Menu                                   4. 主菜单
Enter the option to use:                       输入使用的选项
```

如果需要修改默认设置（端口为 5001，数据包数为 1000，目的 IP 为 192.168.1.9），可在超级终端中输入 1～3，然后按“Enter”（回车）键进行修改。

（10）在超级终端中输入“4”，然后按“Enter”（回车）键回到“选择选项”主菜单。

（11）在超级终端中输入“1”，然后按“Enter”（回车）键选择“发送 TCP 数据包”，超级终端中显示

```
Run iperf command "iperf.exe -s -i 1 -t 100" and press Enter
```

（12）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-s -i 1 -t 100
```

参数说明：

- -s：服务器模式。
- -i 1：带宽报告周期 1 s，可以省略。
- -t 100：传输时间 100 s，可以省略。

点击“关”按钮，按钮变为“开”：打开服务器模式，同时显示下列信息。

```
------------------------------------------------------------------------
Server listening on TCP port 5001              服务器监听 TCP 端口 5001
TCP windows size: 1.00 MByte(default)          TCP 窗口大小：1 MB（默认）
------------------------------------------------------------------------
```

（13）在超级终端中按“Enter”（回车）键，超级终端中显示

```
Sending TCP packets...                         发送 TCP 数据包
```

安卓手机的 iPerf 应用程序窗口显示

```
[  8] local 192.168.1.9 port 5001 connected with 192.168.1.6 port 64224
[  8] 本地 192.168.1.9 端口 5001 与 192.168.1.6 端口 64224 连接
```

注意：192.168.1.6 的端口号 64224 由系统指定，实际情况可能不同。

数据包发送完成后，超级终端中显示

```
Sent 1000 packets successfully                    发送 1000 数据包成功
```

安卓手机的 iPerf 应用程序窗口显示

```
[ ID] interval    Transfer    Bandwidth
[  8] 0.0-2.9 sec 1.34 Mbytes 3.87 Mbits/sec
```

（14）在超级终端中输入“2”，然后按“Enter”（回车）键选择“接收 TCP 数据包”，超级终端中显示

```
Press Enter and run iperf command "iperf.exe -c 192.168.1.6 -i 1 -t 100000"
```

（15）在超级终端中按“Enter”（回车）键，超级终端中显示

```
Receiving TCP packets...                          接收 TCP 数据包
```

（16）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-c 192.168.1.6 -i 1 -t 100000
```

参数说明：

- -c 192.168.1.6：客户端模式，连接到 192.168.1.6（服务器 IP 地址）。
- -i 1：带宽报告周期 1 s，可以省略。
- -t 100000：传输时间 100 000 s，可以省略。

单击“开”按钮，按钮变为“关”，表示关闭服务器模式；再单击“关”按钮，按钮变为“开”，表示打开客户端模式，同时显示下列信息。

```
------------------------------------------------------------------------
Client connecting to 192.168.1.6, TCP port 5001
                                        客户端连接 192.168.1.6，TCP 端口 5001
TCP windows size: 512 KByte(default)    TCP 窗口大小：512 KB（默认）
------------------------------------------------------------------------
[  7] local 192.168.1.9 port 60751 connected with 192.168.1.6 port 5001
[  7] 本地 192.168.1.9 端口 60751 与 192.168.1.6 端口 5001 连接
[ ID] interval    Transfer    Bandwidth
[  7] 0.0-8.6 sec 1.25 Mbytes 1.22 Mbits/sec
```

客户端数据包发送完成后，“开”按钮变为“关”，表示关闭客户端模式。超级终端中显示：

```
Recieved 1000 packets successfully                接收 1000 数据包成功
```

（17）在超级终端中输入“4”，然后按“Enter”（回车）键选择“退出”，退出主菜单。

（18）单击调试工具栏中的“Suspend”（暂停）按钮，停止程序的运行。

（19）单击调试工具栏中的“Restart”（重新开始）按钮，为程序的分析做准备。

下面重点介绍程序中 BsdTcpServer()和 BsdTcpClient()的分析。

6.1.1 BsdTcpServer()分析

BsdTcpServer()的分析步骤如下。

（1）在 main.c 中 BsdTcpServer()函数的下列语句设置断点。

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
```

（2）单击调试工具栏中的“Resume”（运行）按钮，超级终端中显示“选择选项”主菜单。

（3）在超级终端中输入“2”，然后按“Enter”（回车）键选择“接收 TCP 数据包”，超级终端中显示下列提示信息。

```
Press Enter and run iperf command "iperf.exe -c 192.168.1.6 -i 1 -t 100000"
按回车键，运行 iperf 命令
iperf.exe -c 192.168.1.6 -i 1 -t 100000
```

（4）在超级终端中按“Enter”（回车）键，超级终端中显示：

```
Receiving TCP packets...                    接收 TCP 数据包
```

程序停在下列语句：

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
```

TCP 服务器函数 BsdTcpServer()包括下列内容。

- 创建 TCP 套接字：sl_Socket()。
- 绑定 TCP 服务器：sl_Bind()。
- 监听 TCP 连接：sl_Listen()。
- 设置非阻塞模式：sl_SetSockOpt()。
- 接受 TCP 连接：sl_Accept()。
- 接收 TCP 数据包：sl_Recv()。
- 关闭 TCP 套接字：sl_Close()。

BsdTcpServer()运行时，在指定端口监听，一旦有客户端连接，就可以收发数据。

1. 创建 TCP 套接字 sl_Socket()

创建 TCP 套接字的函数是 sl_Socket()，函数格式为

```
_i16 sl_Socket(_i16 Domain, _i16 Type, _i16 Protocol);
```

功能：创建套接字。

参数说明：

- Domain：协议簇，传递值是 SL_AF_INET（IPv4），在 socket.h 中定义为 2。
- Type：类型，传递值是 SL_SOCK_STREAM（流式套接字），在 socket.h 中为 1。
- Protocol：传递值是 0。

返回值：正值表示套接字 ID，负值表示错误代码。

sl_Socket()的声明在 socket.h 中，代码在 socket.c 中。

2．绑定 TCP 服务器 sl_Bind()

绑定 TCP 服务器的函数是 sl_Bind()，调用语句为

```
iStatus = sl_Bind(iSockID, (SlSockAddr_t *)&sLocalAddr, iAddrSize);
```

sl_Bind()函数格式为

```
_i16 sl_Bind(_i16 sd, const SlSockAddr_t *addr, _i16 addrlen);
```

功能：绑定套接字到服务器地址。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- addr：服务器地址结构指针，服务器地址是 0.0.0.0（自身地址）。
- addrlen：服务器地址结构长度，传递值是 16。

返回值：0 表示成功，负值表示错误代码。

sl_Bind()的声明在 socket.h 中，代码在 socket.c 中。

3．监听 TCP 连接 sl_Listen()

监听 TCP 连接的函数是 sl_Listen()，调用语句为

```
iStatus = sl_Listen(iSockID, 0);
```

sl_Listen()函数格式为

```
_i16 sl_Listen(_i16 sd, _i16 backlog);
```

功能：监听 TCP 连接。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- backlog：监听队列深度，传递值是 0。

返回值：0 表示成功，负值表示错误代码。

sl_Listen()的声明在 socket.h 中，代码在 socket.c 中。

4．设置非阻塞模式 sl_SetSockOpt()

设置非阻塞模式的函数是 sl_SetSockOpt()，调用语句为

```
iStatus = sl_SetSockOpt(iSockID, SL_SOL_SOCKET, SL_SO_NONBLOCKING,
                        &lNonBlocking, sizeof(lNonBlocking));
```

sl_SetSockOpt()函数格式为

```
_i16 sl_SetSockOpt(_i16 sd, _i16 level, _i16 optname, const void *optval,
                   SlSocklen_t optlen);
```

功能：设置套接字选项。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- level：选项类别，传递值是 SL_SOL_SOCKET（套接字选项），在 socket.h 中为 1。
- optname：选项名称，传递值是 SL_SO_NONBLOCKING（非阻塞模式），在 socket.h 中定义为 24。非阻塞模式时 sl_Accept()、sl_Connect()和 sl_Recv()等操作不阻塞。
- optval：选项值，传递值是 1，表示允许非阻塞模式。
- optlen：选项值长度，传递值是 4。

返回值：0 表示成功，负值表示错误代码。

sl_SetSockOpt()的声明在 socket.h 中，代码在 socket.c 中。

注意：套接字的默认状态是阻塞模式，即 sl_Accept()、sl_Connect()和 sl_Recv()等操作一直等待操作完成后才返回。这对 sl_Accept()和 sl_Connect()是必须的，因为如果连接不建立，就无法完成数据的发送和接收。但对 sl_Recv()，通常使用非阻塞模式工作。

BsdTcpServer()中设置非阻塞模式后，sl_Accept()操作使用“重试”方法实现阻塞，如图 6.4 所示。

```
982     iNewSockID = SL_EAGAIN;
983
984     // waiting for an incoming TCP connection
985     while( iNewSockID < 0 )
986     {
987         // accepts a connection form a TCP client, if there is any
988         // otherwise returns SL_EAGAIN
989         iNewSockID = sl_Accept(iSockID, ( struct SlSockAddr_t *)&sAddr,
990                                (SlSocklen_t*)&iAddrSize);
991         if( iNewSockID == SL_EAGAIN )
992         {
993            MAP_UtilsDelay(10000);
994         }
995         else if( iNewSockID < 0 )
996         {
997             // error
998             sl_Close(iNewSockID);
999             sl_Close(iSockID);
1000            ASSERT_ON_ERROR(ACCEPT_ERROR);
1001        }
1002    }
```

图 6.4　Tcp_socket 项目 BsdTcpServer()函数（部分）

5．接受 TCP 连接 sl_Accept()

接受 TCP 连接的函数是 sl_Accept()，调用语句为

```
iNewSockID = sl_Accept(iSockID, (struct SlSockAddr_t *)&sAddr,
                       (SlSocklen_t*)&iAddrSize);
```

sl_Accept()函数格式为

```
_i16 sl_Accept(_i16 sd, SlSockAddr_t *addr, SlSocklen_t *addrlen);
```

功能：接受 TCP 连接。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- sAddr：客户端地址结构指针，用于存放返回的客户端地址（192.168.1.9）。
- iAddrSize：客户端地址结构长度，传递值是 16。

返回值：正值表示套接字 ID，负值表示错误代码，非阻塞模式可能返回 SL_EAGAIN（重试）。

sl_Accept()的声明在 socket.h 中，代码在 socket.c 中。sl_Accept()接受客户端 TCP 连接后，新建 TCP 套接字用于收发数据。

6．接收 TCP 数据包 sl_Recv()

接收 TCP 数据包的函数是 sl_Recv()，函数格式为

```
_i16 sl_Recv(_i16 sd, void *pBuf, _i16 Len, _i16 flags);
```

功能：接收 TCP 数据包。

参数说明：

- sd：套接字 ID，传递值是 sl_Accept()的返回值。
- pBuf：数据包指针，用于存放接收的数据包。
- Len：数据包字节数（1～16000），传递值是 1400。
- flags：数据包类型，传递值是 0。

返回值：正值表示接收字节数，负值表示错误代码，非阻塞模式可能返回 SL_EAGAIN（重试）。

sl_Recv()的声明在 socket.h 中，代码在 socket.c 中。

7．关闭 TCP 套接字 sl_Close()

关闭 TCP 套接字的函数是 sl_Close()，函数格式为

```
_i16 sl_Close(_i16 sd);
```

功能：关闭套接字。

参数说明：sd，套接字 ID，传递值分别是 sl_Accept()和 sl_Socket()的返回值。

返回值：0 表示成功，负值表示错误代码。

sl_Close()的声明在 socket.h 中，代码在 socket.c 中。

（1）在 main.c 中 BsdTcpServer()函数的下列语句设置断点。

```
return SUCCESS;
```

（2）单击调试工具栏中的“Resume”（运行）按钮，运行 TCP 服务端函数：首先创建 TCP 套接字、绑定 TCP 服务器并监听 TCP 连接，然后等待接受 TCP 连接。

（3）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-c 192.168.1.6 -i 1 -t 100000
```

点击“关”按钮，按钮变为“开”，表示打开客户端模式，同时显示下列信息。

```
------------------------------------------------------------
Client connecting to 192.168.1.6, TCP port 5001
                                    客户端连接 192.168.1.6，TCP 端口 5001
TCP windows size: 512 KByte(default)
                                    TCP 窗口大小：512KB（默认）
------------------------------------------------------------
[  7] local 192.168.1.9 port 51607 connected with 192.168.1.6 port 5001
[  7] 本地 192.168.1.9 端口 51607 连接到 192.168.1.6 端口 5001
```

服务器接受 TCP 连接后，利用新建 TCP 套接字循环接收客户端发送的 1000 个 TCP 数据包。

数据包发送完成后，安卓手机 iPerf 应用程序对话框中的“开”按钮变为“关”：关闭客户端模式，iPerf 应用程序窗口显示

```
[ ID] interval    Transfer    Bandwidth
[  7] 0.0-8.6 sec 1.25 Mbytes 1.22 Mbits/sec
```

服务器关闭两个 TCP 套接字，超级终端中显示

```
Recieved 1000 packets successfully          接收 1000 数据包成功
```

6.1.2 BsdTcpClient()分析

BsdTcpClient()分析步骤如下。

（1）在 main.c 中 BsdTcpClient()函数的下列语句设置断点。

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
```

（2）单击调试工具栏中的“Resume”（运行）按钮，超级终端中显示提示信息和“选择选项”主菜单。

（3）在超级终端中输入“1”，然后按“Enter”（回车）键选择“发送 TCP 数据包”，在超级终端中显示下列提示信息。

```
Run iperf command "iperf.exe -s -i 1 -t 100" and press Enter
运行 iperf 命令“iperf.exe -s -i 1 -t 100”，按回车键
```

（4）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-s -i 1 -t 100
```

点击“关”按钮，按钮变为“开”，表示打开服务器模式，同时显示下列信息。

```
------------------------------------------------------------
Server listening on TCP port 5001           服务器监听 TCP 端口 5001
TCP windows size: 1.00 MByte(default)       TCP 窗口大小：1MB（默认）
------------------------------------------------------------
```

（5）在超级终端中按“Enter”（回车）键，超级终端中显示

```
Sending TCP packets...                      发送 TCP 数据包
```

程序停在下列语句。

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_STREAM, 0);
```

TCP 客户端函数 BsdTcpClient()包括下列内容。

- 创建 TCP 套接字：sl_Socket()。
- 连接 TCP 服务器：sl_Connect()。
- 发送 TCP 数据包：sl_Send()。
- 关闭 TCP 套接字：sl_Close()。

注意：调用 BsdTcpClient()前，必须有服务器在指定端口监听。

1．创建 TCP 套接字 sl_Socket()

创建 TCP 套接字的函数是 sl_Socket()，函数格式为

```
_i16 sl_Socket(_i16 Domain, _i16 Type, _i16 Protocol);
```

功能：创建套接字。

参数说明：

- Domain：协议簇，传递值是 SL_AF_INET（IPv4），在 socket.h 中定义为 2。
- Type：类型，传递值是 SL_SOCK_STREAM（流式套接字或可靠的面向流的服务，TCP）。Type 主要包括下列值。
 - ✧ SL_SOCK_STREAM：流式套接字（TCP），1。
 - ✧ SL_SOCK_DGRAM：数据报套接字（UDP），2。
- Protocol：套接字特定协议，传递值是 0。

返回值：正值表示套接字 ID，负值表示错误代码。

sl_Socket()的声明在 socket.h 中，代码在 socket.c 中。

2．连接 TCP 服务器 sl_Connect()

连接 TCP 服务器的函数是 sl_Connect()，调用语句如图 6.5 所示。

```
872     // connecting to TCP server
873     iStatus = sl_Connect(iSockID, ( SlSockAddr_t *)&sAddr, iAddrSize);
874     if( iStatus < 0 )
875     {
876         // error
877         sl_Close(iSockID);
878         ASSERT_ON_ERROR(CONNECT_ERROR);
879     }
```

图 6.5　Tcp_socket 项目 BsdTcpClient()函数（部分）

sl_Connect()函数格式为

```
_i16 sl_Connect(_i16 sd, const SlSockAddr_t *addr, _i16 addrlen);
```

功能：连接 TCP 服务器。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。

- addr：服务器地址指针，服务器地址是 192.168.1.9（手机 IP 地址）。
- addrlen：地址长度，传递值是 16。

返回值：0 表示成功，负值表示错误代码，非阻塞模式可能返回 SL_EALREADY（未准备好）。

sl_Connect()的声明在 socket.h 中，代码在 socket.c 中。

3．发送 TCP 数据包 sl_Send()

发送 TCP 数据包的函数是 sl_Send()，调用语句是

```
iStatus = sl_Send(iSockID, g_cBsdBuf, sTestBufLen, 0);
```

sl_Send()函数格式为

```
_i16 sl_Send(_i16 sd, const void *pBuf, _i16 Len, _i16 flags);
```

功能：发送 TCP 数据包。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- pBuf：数据包指针，指向发送的数据包。
- Len：数据包字节数（1～1460），传递值是 1400。
- flags：数据包类型，传递值是 0。

返回值：正值表示发送字节数，–1 表示错误。

sl_Send()的声明在 socket.h 中，代码在 socket.c 中。

4．关闭 TCP 套接字

关闭 TCP 套接字的函数是 sl_Close()，函数格式为

```
_i16 sl_Close(_i16 sd);
```

功能：关闭套接字。

参数说明：sd，套接字 ID，传递值是 sl_Socket()的返回值。

返回值：0 表示成功，负值表示错误代码。

sl_Close()的声明在 socket.h 中，代码在 socket.c 中。

（1）在 main.c 中 BsdTcpClient()函数的下列语句设置断点。

```
return SUCCESS;
```

（2）单击调试工具栏中的“Resume”（运行）按钮，运行 TCP 客户端函数。首先创建 TCP 套接字并连接 TCP 服务器，然后循环发送 1000 个 TCP 数据包，最后关闭 TCP 套接字。

第一个数据包发送成功后，安卓手机的 iPerf 应用程序窗口显示

```
[  8] local 192.168.1.9 port 5001 connected with 192.168.1.6 port 49873
[  8] 本地 192.168.1.9 端口 5001 与 192.168.1.6 端口 49873 连接
```

所有数据包都发送成功后，在超级终端中显示

```
Sent 1000 packets successfully                    发送 1000 数据包成功
```

关闭 TCP 套接字后，安卓手机的 iPerf 应用程序窗口显示

```
[ ID] interval    Transfer    Bandwidth
[  8] 0.0-7.1 sec 1.34 Mbytes 582 Kbits/sec
```

综上所述，Tcp_socket 项目使用的 SimpleLink API 函数如表 6.1 所示。

表 6.1　Tcp_socket 项目使用的 SimpleLink API 函数

功　能	函数名	调用函数	代码文件
启动 SimpleLink	sl_Start()	main()	device.c
连接 WLAN	sl_WlanConnect()	WlanConnect()	wlan.c
创建套接字	**sl_Socket()**	BsdTcpClient()、BsdTcpServer()	socket.c
连接服务器	**sl_Connect()**	BsdTcpClient()	socket.c
发送 TCP 数据包	**sl_Send()**	BsdTcpClient()	socket.c
绑定服务器	**sl_Bind()**	BsdTcpServer()	socket.c
监听 TCP 连接	**sl_Listen()**	BsdTcpServer()	socket.c
设置套接字选项	sl_SetSockOpt()	BsdTcpServer()	socket.c
接受 TCP 连接	**sl_Accept()**	BsdTcpServer()	socket.c
接收 TCP 数据包	**sl_Recv()**	BsdTcpServer()	socket.c
关闭套接字	**sl_Close()**	BsdTcpClient()、BsdTcpServer()	socket.c
停止 SimpleLink	sl_Stop()	main()	device.c

6.2　Udp_socket 示例程序分析

Udp_socket 示例程序在“example\udp_socket”目录中，Udp_socket 项目的工作示意图如图 6.6 所示。



图 6.6　Udp_socket 项目工作示意图

CC3200 站点首先连接接入点，然后通过接入点与连接到接入点的 Wi-Fi 站点建立 UDP 套接字，最后通过 UDP 套接字实现 UDP 数据包的发送和接收。

Udp_socket 项目包含下列 5 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- udma_if.c：DMA 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Udp_socket 项目程序流程如图 6.7 所示。

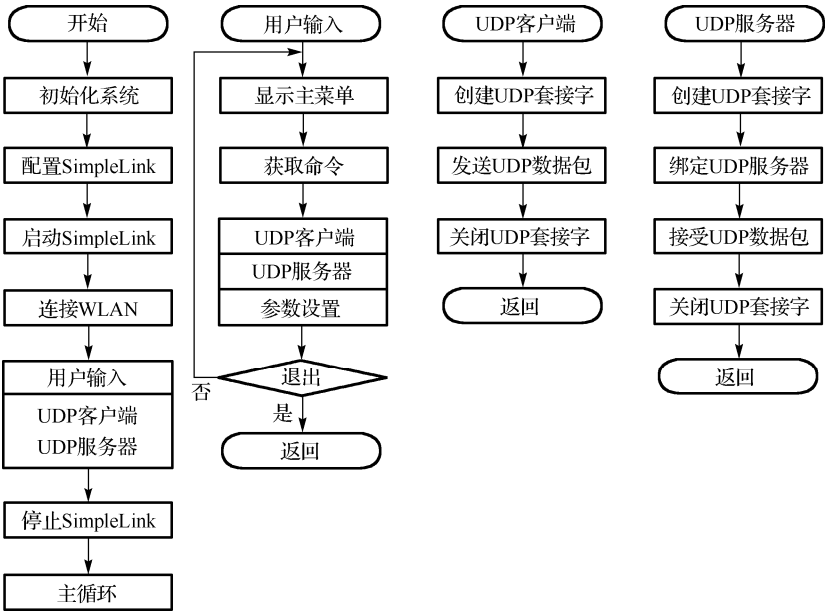


图 6.7　Udp_socket 项目程序流程

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）在安卓手机（或笔记本电脑）上打开 Wi-Fi 并连接到用户接入点（如 1234），查看接入点分配给手机的 IP 地址（如 192.168.1.9）。

（2）在安卓手机上安装并运行 iperf.apk。

（3）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（4）在 PC 上打开超级终端。

（5）将 main.c 中的下列宏定义。

```
#define IP_ADDR         0xc0a8006E          /* 192.168.0.110 */
```

修改为接入点分配给手机的 IP 地址，例如：

```
#define IP_ADDR         0xc0a80109          /* 192.168.1.9 */
```

（6）将 common.h 中的下列宏定义

```
#define SSID_NAME       "cc3200demo"        /* 接入点名称 */
#define SECURITY_TYPE   SL_SEC_TYPE_OPEN    /* 安全类型(OPEN/WEP/WPA) */
#define SECURITY_KEY    ""                  /* 安全密码 */
```

修改为自己 AP 的配置值，例如：

```
#define SSID_NAME       "1234"              /* 接入点名称 */
#define SECURITY_TYPE   SL_SEC_TYPE_WPA     /* 安全类型(OPEN/WEP/WPA) */
#define SECURITY_KEY    "1234asdf"          /* 安全密码 */
```

注意：如果上述宏定义在 WLAN 站点示例程序分析中已经修改，此步可以省略。

（7）选中 udp_socket 项目，按“F11”键，CCS 生成输出文件 udp_socket.out，并将 udp_socket.out 下载到开发板的 CC3200 中，然后进入调试界面。

（8）单击调试工具栏中的“Resume”（运行）按钮，运行程序，超级终端中显示下列信息。

```
*********************************************************************
                    CC3200 UDP Socket Application
*********************************************************************

Host Driver Version: 1.0.0.10
Build Version 2.4.0.2.31.1.3.0.1.1.0.3.34
Device is configured in default state
Device started as STATION
Connecting to AP: 1234 ...                    连接接入点：1234
[WLAN EVENT] STA Connected to the AP: 1234 , BSSID: 3c:e5:a6:1:7:bc
[NETAPP EVENT] IP Acquired: IP=192.168.1.6 , Gateway=192.168.1.1
Connected to AP: 1234                         连接到接入点：1234
Device IP: 192.168.1.6

Default settings: SSID Name: 1234, PORT = 5001, Packet Count = 1000,
                  Destination IP: 192.168.1.9

Options:                                      选项
1. Send UDP packets.                          1. 发送 UDP 数据包
2. Receive UDP packets.                       2. 接收 UDP 数据包
3. Settings.                                  3. 设置
4. Exit                                       4. 退出
Enter the option to use:                      输入使用的选项
```

（9）在超级终端中输入“1”，然后按“Enter”（回车）键选择“发送 UDP 数据包”，在超级终端中显示

```
Run iperf command "iperf.exe -u -s -i 1" and press Enter
```

（10）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-u -s -i 1
```

参数说明：

- -u：UDP 连接。
- -s：服务器模式。
- -i 1：带宽报告周期 1 s，可以省略。

点击“关”按钮，按钮变为“开”，表示打开服务器模式，同时显示下列信息。

```
------------------------------------------------------------
Server listening on UDP port 5001             服务器监听 UDP 端口 5001
Receiving 1470 byte datagrams                 接收 1 470 B 数据报
```

```
UDP buffer size: 110 KByte(default)        UDP 缓存大小：110 KB（默认）
------------------------------------------------------------------------
```

（11）在超级终端中按“Enter”（回车）键，在超级终端中显示

```
Sending UDP packets...                     发送 UDP 数据包
```

安卓手机的 iPerf 应用程序窗口显示

```
[  7] local 192.168.1.9 port 5001 connected with 192.168.1.6 port 58146
[  7] 本地 192.168.1.9 端口 5001 与 192.168.1.6 端口 58146 连接
[ ID] interval    Transfer    Bandwidth      Jitter    Lost/Total Datagrams
[  7] 0.0-1.0 sec 107 Kbytes 874 Kbits/sec 12.310 ms 65643/66051(99%)
```

数据包发送完成后，在超级终端中显示

```
Sent 1000 packets successfully             发送 1000 数据包成功
```

（12）在超级终端中输入“2”，然后按“Enter”（回车）键选择“接收 TCP 数据包”，在超级终端中显示

```
Run iperf command "iperf.exe -u -c 192.168.1.6 -i 1 -t 100000" and press Enter
```

（13）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-u -c 192.168.1.6 -i 1 -t 100000
```

参数说明：

- -u：UDP 连接。
- -c 192.168.1.6：客户端模式，连接到 192.168.1.6（服务器 IP 地址）。
- -i 1：带宽报告周期 1 s，可以省略。
- -t 100000：传输时间 100 000 s，可以省略。

单击“开”按钮，按钮变为“关”，表示关闭服务器模式；再单击“关”按钮，按钮变为“开”，表示打开客户端模式，同时显示下列信息。

```
------------------------------------------------------------------------
Client connecting to 192.168.1.6, UDP port 5001
                                           客户端连接 192.168.1.6,UDP 端口 5001
Sending 1470 byte datagrams                发送 1 470 B 数据报
UDP buffer size: 110 KByte(default)        UDP 缓存大小：110 KB（默认）
------------------------------------------------------------------------
[  7] local 192.168.1.9 port 44962 connected with 192.168.1.6 port 5001
[  7] 本地 192.168.1.9 端口 44962 与 192.168.1.6 端口 5001 连接
[ ID] interval    Transfer    Bandwidth
[  7] 0.0-1.0 sec 129 Kbytes 1.06 Mbits/sec
```

（14）在超级终端中按“Enter”（回车）键，在超级终端中显示

```
Receiving UDP packets...                   接收 UDP 数据包
```

数据包接收完成后，在超级终端中显示下列信息。

```
Recieved 1000 packets successfully         接收 1000 数据包成功
```

单击“开”按钮，按钮变为“关”，表示关闭客户端模式。

（15）在超级终端中输入“4”，然后按“Enter”（回车）键选择“退出”，退出主菜单。

（16）单击调试工具栏中的“Suspend”（暂停）按钮，停止程序的运行。

（17）单击调试工具栏中的“Restart”（重新开始）按钮，为程序的分析做准备。

下面重点介绍程序中 BsdUdpServer()和 BsdUdpClient()的分析。

6.2.1　BsdUdpServer()分析

BsdUdpServer()的分析步骤如下。

（1）在 main.c 中 BsdUdpServer()函数的下列语句设置断点。

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_DGRAM, 0);
```

（2）单击调试工具栏中的“Resume”（运行）按钮，在超级终端中显示“选项”主菜单。

（3）在超级终端中输入“2”，然后按“Enter”（回车）键选择“接收 UDP 数据包”，在超级终端中显示下列提示信息。

```
Run iperf command "iperf.exe -u -c 192.168.1.6 -i 1 -t 100000" and press Enter
运行 iperf 命令“iperf.exe -u -c 192.168.1.6 -i 1 -t 100000”，按回车键
```

（4）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-u -c 192.168.1.6 -i 1 -t 100000
```

点击“关”按钮，按钮变为“开”，表示打开客户端模式，同时显示下列信息。

```
------------------------------------------------------------
Client connecting to 192.168.1.6, UDP port 5001
                                    客户端连接 192.168.1.6，UDP 端口 5001
Sending 1470 byte datagrams             发送 1 470 B 数据报
UDP buffer size: 110 KByte(default)     UDP 缓存大小：110 KB（默认）
------------------------------------------------------------
[  7] local 192.168.1.9 port 52040 connected with 192.168.1.6 port 5001
[  7] 本地 192.168.1.9 端口 52040 连接到 192.168.1.6 端口 5001
[ ID] interval    Transfer    Bandwidth
[  7] 0.0-1.0 sec 129 Kbytes 1.06 Mbits/sec
```

（5）在超级终端中按“Enter”（回车）键，在超级终端中显示

```
Receiving UDP packets...                接收 UDP 数据包
```

程序停在下列语句。

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_DGRAM, 0);
```

UDP 服务器函数 BsdUdpServer()包括下列内容。

- 创建 UDP 套接字：sl_Socket()。

- 绑定 UDP 服务器：sl_Bind()。
- 接收 UDP 数据包：sl_RecvFrom()。
- 关闭 UDP 套接字：sl_Close()。

1．创建 UDP 套接字 sl_Socket()

创建 UDP 套接字的函数是 sl_Socket()，函数格式为

```
_i16 sl_Socket(_i16 Domain, _i16 Type, _i16 Protocol);
```

功能：创建套接字。

参数说明：

- Domain：协议簇，传递值是 SL_AF_INET（IPv4），在 socket.h 中定义为 2。
- Type：类型，传递值是 SL_SOCK_DGRAM（数据报套接字或数据报服务，UDP），在 socket.h 中定义为 2。
- Protocol：套接字特定协议，传递值是 0。

返回值：正值表示套接字 ID，负值表示错误代码。

sl_Socket()的声明在 socket.h 中，代码在 socket.c 中。

2．绑定 UDP 服务器 sl_Bind()

绑定 UDP 服务器的函数是 sl_Bind()，调用语句为

```
iStatus = sl_Bind(iSockID, (SlSockAddr_t *)&sLocalAddr, iAddrSize);
```

sl_Bind()函数格式为

```
_i16 sl_Bind(_i16 sd, const SlSockAddr_t *addr, _i16 addrlen);
```

功能：绑定套接字到服务器地址。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- addr：服务器地址结构指针，服务器地址是 0.0.0.0（自身地址）。
- addrlen：服务器地址结构长度，传递值是 16。

返回值：0 表示成功，负值表示错误代码。

sl_Bind()的声明在 socket.h 中，代码在 socket.c 中。

3．接收 UDP 数据包 sl_RecvFrom()

接收 UDP 数据包的函数是 sl_RecvFrom()，调用语句为

```
iStatus = sl_RecvFrom(iSockID, g_cBsdBuf, sTestBufLen, 0,
                      (SlSockAddr_t *)&sAddr, (SlSocklen_t*)&iAddrSize);
```

sl_RecvFrom()函数格式为

```
_i16 sl_RecvFrom(_i16 sd, void *buf, _i16 Len, _i16 flags,
                 SlSockAddr_t *from, SlSocklen_t *fromlen);
```

功能：接收 UDP 数据包。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- buf：数据包指针，用于存放接收的数据包。
- Len：数据包字节数（1～16000），传递值是 1400。
- flags：数据包类型，传递值是 0。
- from：源地址结构指针，用于存放源地址（192.168.1.9）。
- fromlen：源地址结构长度，传递值是 16。

返回值：正值表示发送字节数，–1 表示错误。

sl_RecvFrom()的声明在 socket.h 中，代码在 socket.c 中。

4．关闭 UDP 套接字 sl_Close()

关闭 UDP 套接字的函数是 sl_Close()，函数格式为

```
int sl_Close(int sd);
```

功能：关闭套接字。

参数说明：sd，套接字 ID，传递值是 sl_Socket()的返回值。

返回值：0 表示成功，负值表示错误代码。

sl_Close()的声明在 socket.h 中，代码在 socket.c 中。

（1）在 main.c 中 BsdUdpServer()函数的下列语句设置断点。

```
return SUCCESS;
```

（2）单击调试工具栏中的“Resume”（运行）按钮，运行 UDP 服务端函数，首先创建 UDP 套接字并绑定 UDP 服务器，然后接收客户端发送的数据包。

（3）数据包接收完成后，在超级终端中显示

```
Recieved 1000 packets successfully          接收 1000 数据包成功
```

（4）在安卓手机 iPerf 应用程序的对话框中单击“开”按钮，按钮变为“关”，表示关闭客户端模式。

6.2.2　BsdUdpClient()分析

BsdUdpClient()的分析步骤如下。

（1）在 main.c 中 BsdUdpClient()函数的下列语句设置断点。

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_DGRAM, 0);
```

（2）单击调试工具栏中“Resume”（运行）按钮，超级终端中显示提示信息和主菜单。

（3）在超级终端中输入“1”，然后按“Enter”（回车）键选择“发送 UDP 数据包”，在超级终端中显示下列提示信息。

```
Run iperf command "iperf.exe -u -s -i 1" and press Enter
运行 iperf 命令“iperf.exe -u -s -i 1”，按回车键
```

（4）在安卓手机 iPerf 应用程序的对话框中输入下列 iPerf 命令行参数（iPerf command line arguments）。

```
-u -s -i 1
```

单击“关”按钮，按钮变为“开”，表示打开服务器模式，同时显示下列信息。

```
------------------------------------------------------------
Server listening on UDP port 5001          服务器监听 UDP 端口 5001
Receiving 1470 byte datagrams              接收 1 470 B 数据报
UDP buffer size: 110 KByte(default)        UDP 缓存大小：110 KB（默认）
------------------------------------------------------------
```

（5）在超级终端中按“Enter”（回车）键，在超级终端中显示

```
Sending UDP packets...                     发送 UDP 数据包
```

程序停在下列语句。

```
iSockID = sl_Socket(SL_AF_INET,SL_SOCK_DGRAM, 0);
```

UDP 客户端函数 BsdUdpClient()包括下列内容。

- 创建 UDP 套接字：sl_Socket()。
- 发送 UDP 数据包：sl_SendTo()。
- 关闭 UDP 套接字：sl_Close()。

1．创建 UDP 套接字 sl_Socket()

创建 UDP 套接字的函数是 sl_Socket()，函数格式为

```
_i16 sl_Socket(_i16 Domain, _i16 Type, _i16 Protocol);
```

功能：创建套接字。

参数说明：

- Domain：协议簇，传递值是 SL_AF_INET（IPv4），在 socket.h 中定义为 2。
- Type：类型，传递值是 SL_SOCK_DGRAM（数据报套接字或数据报服务，UDP）。Type 主要包括下列值：
 - ✧ SL_SOCK_STREAM：流式套接字（TCP），1。
 - ✧ SL_SOCK_DGRAM：数据报套接字（UDP），2。
- Protocol：套接字特定协议，传递值是 0。

返回值：正值表示套接字 ID，负值表示错误代码。

sl_Socket()的声明在 socket.h 中，代码在 socket.c 中。

2．发送 UDP 数据包 sl_SendTo()

发送 UDP 数据包的函数是 sl_SendTo()，调用语句为

```
iStatus = sl_SendTo(iSockID, g_cBsdBuf, sTestBufLen, 0,
                    (SlSockAddr_t *)&sAddr, iAddrSize);
```

sl_SendTo()函数格式为

```
_i16 sl_SendTo(_i16 sd, const void *pBuf, _i16 Len, _i16 flags,
               const SlSockAddr_t *to, SlSocklen_t tolen);
```

功能：发送 UDP 数据包。

参数说明：

- sd：套接字 ID，传递值是 sl_Socket()的返回值。
- pBuf：数据包指针，指向发送的数据包。
- Len：数据包字节数（1～1460），传递值是 1400。
- flags：数据包类型，传递值是 0。
- to：目标地址结构指针，目标地址是 192.168.1.9（手机 IP 地址）。
- tolen：目标地址结构长度，传递值是 16。

返回值：正值表示发送字节数，–1 表示错误。

sl_SendTo()的声明在 socket.h 中，代码在 socket.c 中。

3．关闭 UDP 套接字 sl_Close()

关闭 UDP 套接字的函数是 sl_Close()，函数格式为

```
_i16 sl_Close(_i16 sd);
```

功能：关闭套接字。

参数说明：sd，套接字 ID，传递值是 sl_Socket()的返回值。

返回值：0 表示成功，负值表示错误代码。

sl_Close()的声明在 socket.h 中，代码在 socket.c 中。

（1）在 main.c 中 BsdUdpClient()函数的下列语句设置断点。

```
return SUCCESS;
```

（2）单击调试工具栏中“Resume”（运行）按钮，运行 UDP 客户端函数。首先创建 UDP 套接字，然后循环发送 1000 个 UDP 数据包，最后关闭 UDP 套接字。

第一个数据包发送成功后，安卓手机的 iPerf 应用程序窗口显示

```
[  7] local 192.168.1.9 port 5001 connected with 192.168.1.6 port 50960
[  7] 本地 192.168.1.9 端口 5001 与 192.168.1.6 端口 50960 连接
[ ID] interval    Transfer    Bandwidth      Jitter    Lost/Total Datagrams
[  7] 0.0-1.0 sec 319 Kbytes 2.61 Mbits/sec 3.021 ms 65818/66051(1e+02%)
```

所有数据包都发送成功后，在超级终端中显示

```
Sent 1000 packets successfully                发送 1000 数据包成功
```

综上所述，Udp_socket 项目使用的 SimpleLink API 函数如表 6.2 所示。

表 6.2 Udp_socket 项目使用的 SimpleLink API 函数

功　能	函数名	调用函数	代码文件
启动 SimpleLink	sl_Start()	main()	device.c
连接 WLAN	sl_WlanConnect()	WlanConnect()	wlan.c
创建套接字	**sl_Socket()**	BsdUdpClient()、BsdUdpServer()	socket.c
发送 UDP 数据包	**sl_SendTo()**	BsdUdpClient()	socket.c
绑定服务器	**sl_Bind()**	BsdUdpServer()	socket.c
接收 UDP 数据包	**sl_RecvFrom()**	BsdUdpServer()	socket.c
关闭套接字	**sl_Close()**	BsdUdpClient()、BsdUdpServer()	socket.c
停止 SimpleLink	sl_Stop()	main()	device.c

6.3 无线 UART 程序设计

无线 UART 的实现主要包括 WLAN 连接、TCP 套接字连接和无线 UART 传输 3 大步，本地 WLAN 连接又有 3 种方式，如图 6.8 所示。

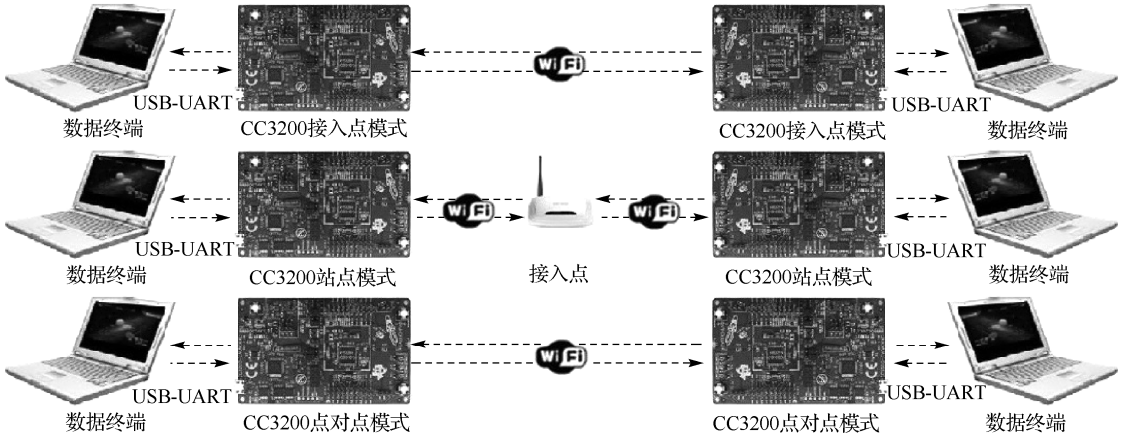


图 6.8 无线 UART 本地 WLAN 连接方式

第 1 种方式，两个 CC3200 分别配置为接入点和站点模式，两者可分别使用 wlan_ap 示例程序和 wlan_station 示例程序实现 WLAN 连接，并与 tcp_socket 示例程序结合实现 TCP 套接字连接和无线 UART 传输。

第 2 种方式，两个 CC3200 均配置为站点模式，两者均可以 Tcp_socket 示例程序为基础，通过接入点实现 WLAN 连接、TCP 套接字连接和无线 UART 传输。

第 3 种方式，两个 CC3200 均配置为点对点模式。

无线 UART 的主要操作是：数据终端通过 USB-UART 转换（或 UART 电平转换）与 CC3200 连接，CC3200 实现 UART/Wi-Fi 转换，并通过 Wi-Fi 实现无线连接。

下面以第 1 种方式为例介绍无线 UART 的程序设计，包括无线 UART 站点 wuart_station 程序设计和无线 UART 接入点 wuart_ap 程序设计。

无线 UART 的工作流程如图 6.9 所示。

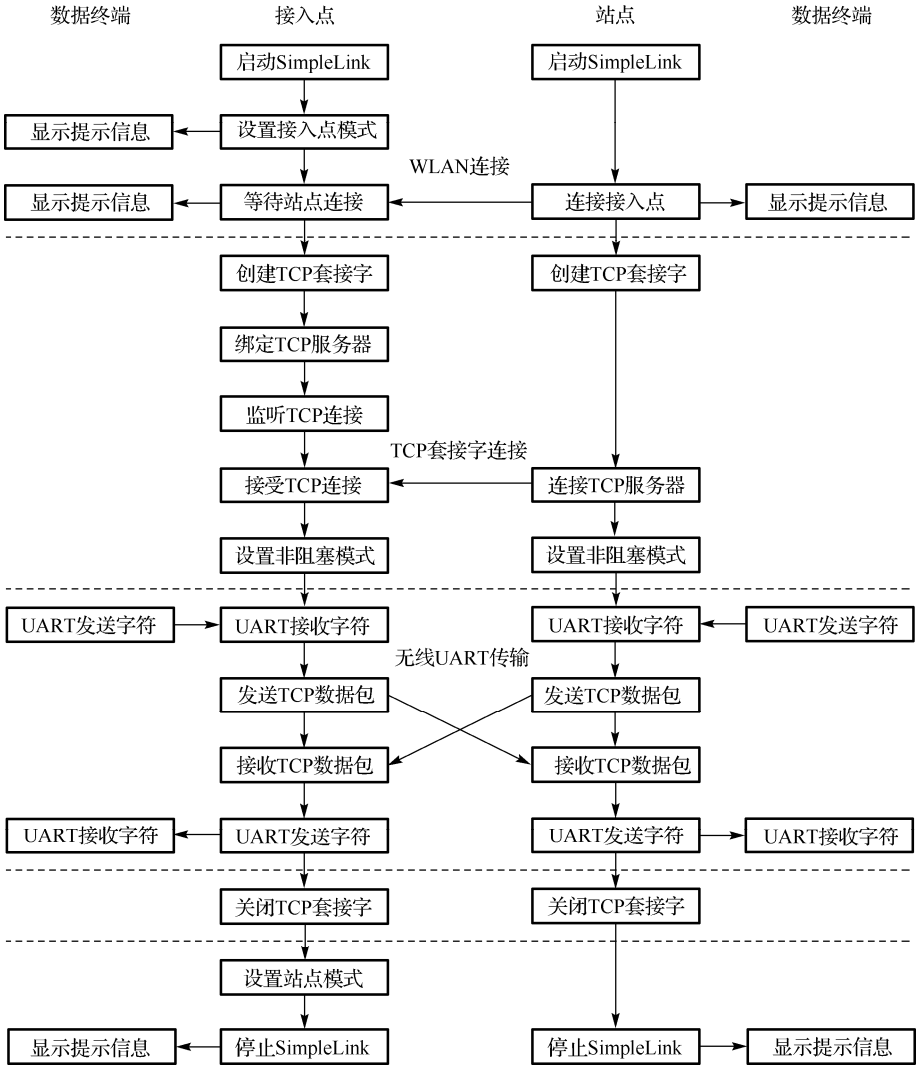


图 6.9　无线 UART 工作流程图

6.3.1　Wuart_station 程序设计

Wuart_station 程序以 wlan_station 示例程序为基础，结合 tcp_socket 示例程序进行设计，具体步骤包括创建 wuart_station 项目、修改 WSM 任务和添加无线 UART 传输函数。

1．创建 wuart_station 项目

为了方便设计，Wuart_station 项目的创建通过修改 wlan_station 项目实现。

① 在“Project Explorer”（项目浏览器）中右击“wlan_station”项目名，在弹出菜单中单击“Rename…”（重命名），将项目 wlan_station 重命名为 wuart_station。

② 在 main.c 中添加下列头文件（用于 UART 操作）。

```
#include "hw_memmap.h"
```

```
#include "uart.h"
```

③ 将 main.c 中的下列宏定义

```
#define APP_NAME              "WLAN STATION"
#define APPLICATION_VERSION   "1.1.1"
```

修改为

```
#define APP_NAME              "WUART STATION"
#define APPLICATION_VERSION   "1.0.0"
```

④ 将 main.c 中 typedef enum{}e_AppStatusCodes 的下列定义

```
LAN_CONNECTION_FAILED = -0x7D0,
INTERNET_CONNECTION_FAILED = LAN_CONNECTION_FAILED - 1,
DEVICE_NOT_IN_STATION_MODE = INTERNET_CONNECTION_FAILED - 1,
```

替换为 tcp_socket 项目中的下列定义。

```
SOCKET_CREATE_ERROR = -0x7D0,
BIND_ERROR = SOCKET_CREATE_ERROR - 1,
LISTEN_ERROR = BIND_ERROR -1,
SOCKET_OPT_ERROR = LISTEN_ERROR -1,
CONNECT_ERROR = SOCKET_OPT_ERROR -1,
ACCEPT_ERROR = CONNECT_ERROR - 1,
SEND_ERROR = ACCEPT_ERROR -1,
RECV_ERROR = SEND_ERROR -1,
SOCKET_CLOSE_ERROR = RECV_ERROR -1,
DEVICE_NOT_IN_STATION_MODE = SOCKET_CLOSE_ERROR - 1,
```

⑤ 将 main.c 中的下列函数声明注释掉。

```
//static long CheckLanConnection();
//static long CheckInternetConnection();
```

⑥ 将 example\common\commoc.h 中的下列宏定义

```
#define SSID_NAME     "cc3200demo"     /* AP SSID */
#define SECURITY_TYPE SL_SEC_TYPE_OPEN/* Security type (OPEN or WEP or WPA*/
#define SECURITY_KEY ""               /* Password of the secured AP */
```

修改为

```
#define SSID_NAME     "asdf"           /* 接入点名称 */
#define SECURITY_TYPE SL_SEC_TYPE_OPEN /* 安全类型(OPEN/WEP/WPA) */
#define SECURITY_KEY  ""               /* 安全密码 */
```

注意：接入点名称必须和 Wuart_ap 中的设置一致。

2．修改 WSM 任务

修改 WSM 任务包括修改 WlanStationMode()内容和修改 BsdTcpClient()内容。

（1）修改 WlanStationMode()内容。

① 将 WlanStationMode()中的下列语句

```
    UART_PRINT("Pinging...! \n\r");
    //Checking the Lan connection by pinging to AP gateway
    lRetVal = CheckLanConnection();
    if(lRetVal < 0)
    {
       UART_PRINT("Device couldn't ping the gateway \n\r");
       LOOP_FOREVER();
    }

    //Turn on GREEN LED when device gets PING response from AP
    GPIO_IF_LedOn(MCU_EXECUTE_SUCCESS_IND);

    //Checking the internet connection by pinging to external host
    lRetVal = CheckInternetConnection();
    if(lRetVal < 0)
    {
       UART_PRINT("Device couldn't ping the external host \n\r");
       LOOP_FOREVER();
    }

    //Turn on ORAGE LED when device gets PING response from AP
    GPIO_IF_LedOn(MCU_ORANGE_LED_GPIO);

    UART_PRINT("Device pinged both the gateway and the external host \n\r");

    UART_PRINT("WLAN STATION example executed successfully \n\r");
```

替换为（用于 TCP 传输）

```
    lRetVal = BsdTcpClient(5001);
    if(lRetVal < 0)
    {
       UART_PRINT("TCP Server failed\n\r");
    }
```

② 将 SimpleLinkPingReport()函数体注释掉。

③ 将 CheckLanConnection()和 CheckInternetConnection()函数体替换为 tcp_socket 示例程序中的 BsdTcpClient()函数体。

（2）修改 BsdTcpClient()内容。

① 将 BsdTcpClient()中的下列变量声明注释掉。

```
    //int     iCounter;
    //short   sTestBufLen;
    //long    lLoopCount = 0;
```

② 将 BsdTcpClient()中的下列语句注释掉。

```
    /* filling the buffer
    for (iCounter=0 ; iCounter<BUF_SIZE ; iCounter++)
```

```
    {
        g_cBsdBuf[iCounter] = (char)(iCounter % 10);
    }
    iTestBufLen  = BUF_SIZE; */
```

③ 将 BsdTcpClient()中的下列语句

```
    sAddr.sin_addr.s_addr = sl_Htonl((unsigned int)g_ulDestinationIp);
```

替换为（修改目标 IP 地址）

```
    sAddr.sin_addr.s_addr = sl_Htonl((unsigned int)g_ulGatewayIP);
```

④ 将 BsdTcpClient()中的下列语句

```
    //sending multiple packets to the TCP server
    while (lLoopCount < g_ulPacketCount)
    {
        //sending packet
        iStatus = sl_Send(iSockID, g_cBsdBuf, sTestBufLen, 0 );
        if( iStatus <= 0 )
        {
            //error
            sl_Close(iSockID);
            ASSERT_ON_ERROR(SEND_ERROR);
        }
        lLoopCount++;
    }
    Report("Sent %u packets successfully\n\r",g_ulPacketCount);
```

替换为（用于无线 UART 传输）

```
    //设置非阻塞模式
    long lNonBlocking = 1;
    iStatus = sl_SetSockOpt(iSockID, SL_SOL_SOCKET,
        SL_SO_NONBLOCKING, &lNonBlocking, sizeof(lNonBlocking));
    if( iStatus < 0 )
    {
        sl_Close(iSockID);
        UART_PRINT("TCP Client failed\n\r");
    }

    //无线 UART 传输
    iStatus = WuartTransfer(UARTA0_BASE, iSockID);
    if(iStatus < 0)
    {
        UART_PRINT("WUART Transfer failed\n\r");
    }
```

3．添加无线 UART 传输函数

在 BsdTcpClient()前添加下列无线 UART 传输函数。

```
int WuartTransfer(unsigned long ulBase, int iSockID)
{
    char          cTxBuf[100];
    char          cRxBuf[100];
    char          cGetChar;
    int           iStatus;
    int           iCounter = 0;

    UART_PRINT("\r\n 发送: ");
    while(1)
    {   //UART 接收字符
        cGetChar = MAP_UARTCharGetNonBlocking(ulBase);
        if(cGetChar != 0xff)
        {   //UART 回显字符
            MAP_UARTCharPut(ulBase, cGetChar);
            //保存接收字符
            cTxBuf[iCounter++] = cGetChar;
            //Enter（回车）键（0x0d）或 Esc（退出）键（0x1b）
            if((cGetChar == 0x0d) || (cGetChar == 0x1b))
            {   //回显并保存“换行”(0x0a)
                MAP_UARTCharPut(ulBase, 0x0a);
                cTxBuf[iCounter++] = 0x0a;
                //发送 TCP 数据包
                iStatus = sl_Send(iSockID, cTxBuf, iCounter, 0);
                if(iStatus <= 0)
                {   //错误处理
                    ASSERT_ON_ERROR(sl_Close(iSockID));
                    UART_PRINT("发送数据失败\n\r");
                    break;
                }
                //Enter（回车）键（0x0d）
                if(cGetChar == 0x0d)
                {
                    UART_PRINT("发送: ");
                    iCounter = 0;
                }
                //Esc（退出）键（0x1b）,退出循环
                else
                    break;
            }
        }
        //接收 TCP 数据包
        iStatus = sl_Recv(iSockID, cRxBuf, 100, 0);
        if( iStatus > 0)
        {   //Enter（回车）键（0x0d）
            if(cRxBuf[iStatus-2] == 0x0d)
            {   //保存“字符串结束”(0x00)
```

```
                cRxBuf[iStatus]=0;
                UART_PRINT("\r\n 接收: ");
                //发送字符串
                Message(cRxBuf);
                UART_PRINT("发送: ");
            }
            //Esc（退出）键（0x1b）,退出循环
            else
                break;
        }
    }
    return(iStatus);
}
```

6.3.2 Wuart_ap 程序设计

Wuart_ap 程序以 wlan_ap 示例程序为基础，结合 tcp_socket 示例程序进行设计，具体步骤包括创建 wuart_ap 项目、修改 WAM 任务和添加无线 UART 传输任务。

1. 创建 wuart_ap 项目

为了方便设计，Wuart_ap 项目的创建通过修改 wlan_ap 项目实现。

① 在“Project Explorer”（项目浏览器）中右击“wlan_ap”项目名，在弹出菜单中单击“Rename…”（重命名），将项目 wlan_ap 重命名为 wuart_ap。

② 在 main.c 中添加下列头文件（用于 UART 操作）。

```
#include "hw_memmap.h"
#include "uart.h"
```

③ 将 main.c 中的下列宏定义

```
#define APP_NAME              "WLAN AP"
#define APPLICATION_VERSION   "1.1.1"
```

修改为

```
#define APP_NAME              "WUART AP"
#define APPLICATION_VERSION   "1.0.0"
```

④ 将 main.c 中 typedef enum{}e_AppStatusCodes 的下列定义

```
LAN_CONNECTION_FAILED = -0x7D0,
CLIENT_CONNECTION_FAILED = LAN_CONNECTION_FAILED - 1,
DEVICE_NOT_IN_STATION_MODE = CLIENT_CONNECTION_FAILED - 1,
```

替换为 tcp_socket 项目中的下列定义。

```
SOCKET_CREATE_ERROR = -0x7D0,
BIND_ERROR = SOCKET_CREATE_ERROR - 1,
LISTEN_ERROR = BIND_ERROR -1,
SOCKET_OPT_ERROR = LISTEN_ERROR -1,
```

```
CONNECT_ERROR = SOCKET_OPT_ERROR -1,
ACCEPT_ERROR = CONNECT_ERROR - 1,
SEND_ERROR = ACCEPT_ERROR -1,
RECV_ERROR = SEND_ERROR -1,
SOCKET_CLOSE_ERROR = RECV_ERROR -1,
DEVICE_NOT_IN_STATION_MODE = SOCKET_CLOSE_ERROR - 1,
```

⑤ 在 main.c 中添加下列全局变量（用于任务间参数传递）。

```
int    iNewSockID = 0;
int    iFlag = 1;
```

⑥ 将 main.c 中的下列函数声明注释掉。

```
//static int PingTest(unsigned long ulIpAddr);
//static int GetSsidName(char *pcSsidName, unsigned int uiMaxLen);
```

2．修改 WAM 任务

修改 WAM 任务包括修改 ConfigureMode()内容、修改 WlanAPMode()内容和修改 BsdTcpServer()内容。

（1）修改 ConfigureMode()内容。

① 将 ConfigureMode()中的下列语句

```
char    pcSsidName[33];
```

修改为（定义接入点名称）

```
char    pcSsidName[33] = "asdf";
```

注意：接入点名称必须和 Wuart_station 中的设置一致。

② 将 ConfigureMode()中的下列语句注释掉。

```
//UART_PRINT("Enter the AP SSID name: ");
//GetSsidName(pcSsidName, 33);
```

③ 将 GetSsidName()函数体注释掉。

④ 将 ConfigureMode()中的下列语句

```
UART_PRINT("Device is configured in AP mode\n\r");
```

修改为（显示接入点名称）

```
UART_PRINT("Device is configured in AP mode: %s\n\r", pcSsidName);
```

（2）修改 WlanAPMode()内容。

① 将 WlanAPMode()中的下列变量声明注释掉。

```
//int iTestResult = 0;
```

② 将 WlanAPMode()中的下列语句

```
iTestResult = PingTest(g_ulStaIp);
if(iTestResult < 0)
{
```

```
        UART_PRINT("Ping to client failed \n\r");
    }
    UNUSED(ucDHCP);
    UNUSED(iTestResult);
```

替换为（用于 TCP 传输）

```
    lRetVal = BsdTcpServer(5001);
    if(lRetVal < 0)
    {
        UART_PRINT("TCP Server failed\n\r");
    }
```

③ 将 WlanAPMode()中的下列语句注释掉。

```
    //UART_PRINT("WLAN AP example executed successfully");
```

④ 将 PingTest()函数体替换为 tcp_socket 示例程序中的 BsdTcpServer()函数体。

（3）修改 BsdTcpServer()内容。

① 将 BsdTcpServer()中的下列变量声明注释掉。

```
    //int    iCounter;
    //int    iNewSockID;
    //long   lLoopCount = 0;
    //int    iTestBufLen;
```

② 将 BsdTcpServer()中的下列语句注释掉。

```
    /* filling the buffer
    for (iCounter=0 ; iCounter<BUF_SIZE ; iCounter++)
    {
        g_cBsdBuf[iCounter] = (char)(iCounter % 10);
    }
    iTestBufLen  = BUF_SIZE; */
```

③ 将 BsdTcpServer()中的下列语句

```
    //waits for 1000 packets from the connected TCP client
    while (lLoopCount < g_ulPacketCount)
    {
        iStatus = sl_Recv(iNewSockID, g_cBsdBuf, iTestBufLen, 0);
        if( iStatus <= 0 )
        {
          //error
          sl_Close(iNewSockID);
          sl_Close(iSockID);
          ASSERT_ON_ERROR(RECV_ERROR);
        }
        lLoopCount++;
    }

    Report("Recieved %u packets successfully\n\r", g_ulPacketCount);
```

替换为（任务循环）

```
UART_PRINT("\r\n 发送: ");
while(iFlag)
{
    osi_Sleep(100);
}
```

3．添加无线 UART 传输任务

（1）在 main()中创建 WlanAPMode 任务和启动任务调度语句间添加下列语句。

```
//创建发送任务
lRetVal = osi_TaskCreate(SendTask, "SENDTASK",\
    OSI_STACK_SIZE, NULL, 1, NULL );

//创建接收任务
lRetVal = osi_TaskCreate(ReceiveTask, "RECEIVETASK",\
    OSI_STACK_SIZE,NULL, 1, NULL );
```

（2）在 main.c 中 WlanAPMode 任务函数后添加下列发送任务函数和接收任务函数。

```
//发送任务函数
void SendTask(void *pvParameters)
{
    char    cGetChar;
    char    cTxBuf[100];
    int     iCounter = 0;
    int     iStatus;

    while(1)
    {   //UART 接收字符
        cGetChar = MAP_UARTCharGetNonBlocking(UARTA0_BASE);
        if((cGetChar != 0xff) && (iNewSockID > 0))
        {   //UART 回显字符
            MAP_UARTCharPut(UARTA0_BASE, cGetChar);
            //保存接收字符
            cTxBuf[iCounter++] = cGetChar;
            //Enter（回车）键（0x0d）或 Esc（退出）键（0x1b）
            if((cGetChar == 0x0d) || (cGetChar == 0x1b))
            {   //回显并保存“换行”(0x0a)
                MAP_UARTCharPut(UARTA0_BASE, 0x0a);
                cTxBuf[iCounter++] = 0x0a;
                //发送 TCP 数据包
                iStatus = sl_Send(iNewSockID, cTxBuf, iCounter, 0);
                if((iStatus > 0) && (cGetChar == 0x0d))
                {   //Enter（回车）键（0x0d）
                    UART_PRINT("发送: ");
                    iCounter = 0;
```

```
                }
                else
                {   //发送错误或 Esc（退出）键（0x1b）
                    UART_PRINT("\r 发送数据结束\n\r");
                    iFlag = 0;
                }
            }
        }
        osi_Sleep(100);
    }
}
//接收任务函数
void ReceiveTask(void *pvParameters)
{
    char cRxBuf[100];
    int  iStatus;

    while(1)
    {
        if(iNewSockID > 0)
        {   //接收 TCP 数据包
            iStatus = sl_Recv(iNewSockID, cRxBuf, 100, 0);
            if( iStatus > 0)
            {   //Enter（回车）键（0x0d）
                if(cRxBuf[iStatus-2] == 0x0d)
                {   //保存“字符串结束”（0x00）
                    cRxBuf[iStatus]=0;
                    UART_PRINT("\n\r 接收: ");
                    //发送字符串
                    Message(cRxBuf);
                    UART_PRINT("发送: ");
                }
                else
                {   //Esc（退出）键（0x1b）
                    UART_PRINT("\n\r 接收数据结束\n\r");
                    iFlag = 0;
                }
            }
        }
        osi_Sleep(100);
    }
}
```

6.3.3 将应用程序编程到串行闪存

CCS 生成的输出文件主要是*.out 文件，用于下载到 CC3200 的内部 RAM 中调试运行，系统掉电后将会丢失。

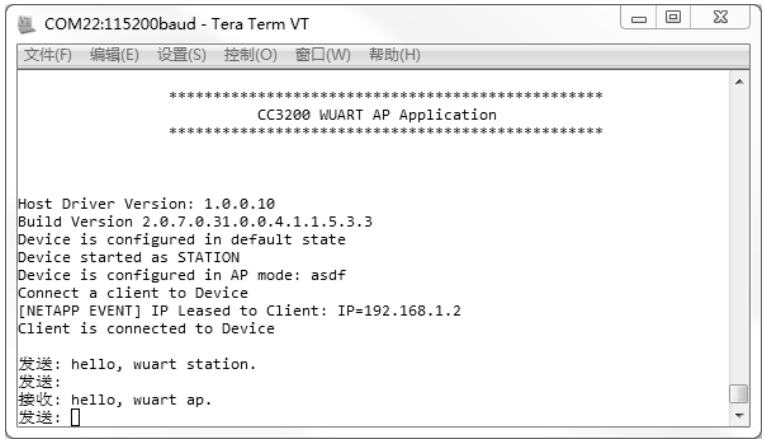
为了在系统加电后运行用户程序，必须使用 UniFlash 将应用程序映像编程到串行闪存中，由启动引导程序将应用程序映像加载到 RAM 中运行。

UniFlash 默认的应用程序映像文件格式是二进制*.bin 文件，使用 UniFlash 将应用程序映像编程到串行闪存的步骤参见 2.2.2 节中服务包更新后半部分的介绍。

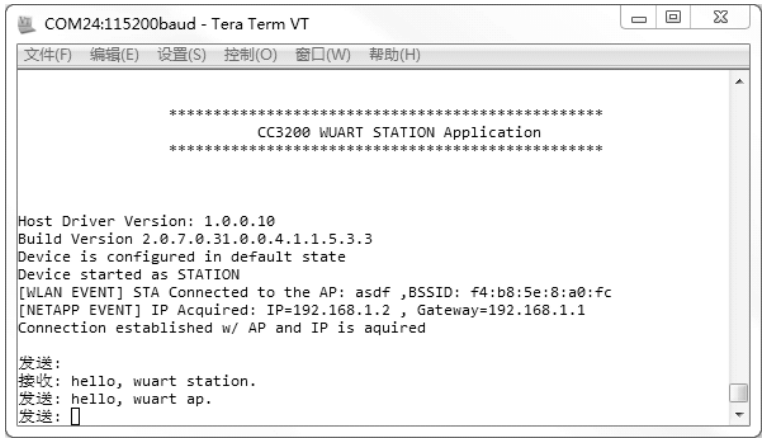
用 UniFlash 分别将下列两个文件编程到两块 CC3200 LaunchPad 的串行闪存中。

```
D:/TI/workspace_v6_1/wuart_ap/Release/wuart_ap.bin
D:/TI/workspace_v6_1/wuart_station/Release/wuart_station.bin
```

编程完成后，拔掉 CC3200 LaunchPad 上连接 J15（1-2）的短路块，用两根 USB 线分别将两块 CC3200 LaunchPad 连接到一台或两台 PC，再分别打开两个超级终端，按一下 CC3200 LaunchPad 上的“RESET”（复位）按键 SW1 开始执行用户程序，在两个超级终端中可以看到相关提示信息，并可进行双向无线 UART 通信，如图 6.10 所示。



(a) Wuart_ap 程序运行结果



(b) Wuart_station 程序运行结果

图 6.10　无线 UART 程序运行结果

第 7 章

网络应用配置

CC3200 配置为站点时，为了实现与接入点的 WLAN 连接，必须配置接入点的名称（SSID）、安全类型和安全密码等参数。

在 5.1 节、6.1 节和 6.2 节中，这些配置参数是通过修改程序中的宏定义实现的，这在实际使用时显然是不可能的。

CC3200 使用配置文件（Profile）解决了这个问题，即事先将接入点的名称（SSID）、安全类型和安全密码等参数以配置文件的形式存放在串行闪存中，CC3200 初始化时从闪存中读取这些配置文件，并利用其中的参数对 CC3200 进行配置。

将配置参数以配置文件的形式存放在串行闪存中的方法有下列 3 种。

- SmartConfig：TI 独特的一步式 Wi-Fi 智能配置技术。
- WPS（Wi-Fi Protected Setup，Wi-Fi 保护设置）：使用 PBC 和 PIN 进行配置。
- AP 模式：在接入点模式下使用浏览器进行配置。

本章通过分析 Provisioning_smartconfig 示例程序和 Provisioning_wps 示例程序介绍 SmartConfig 和 WPS 的使用。AP 模式使用了 HTTP 服务器，将在第 8 章介绍。

7.1 Provisioning_smartconfig 示例程序分析

Provisioning_smartconfig 示例程序在“example\provisioning_smartconfig”目录中，Provisioning_smartconfig 项目的工作示意图如图 7.1 所示。

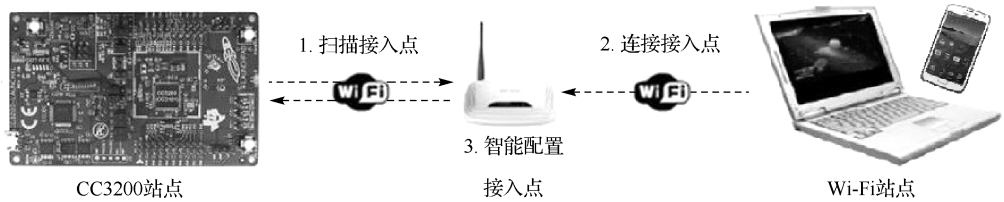


图 7.1 Provisioning_smartconfig 项目工作示意图

CC3200 站点运行启动智能配置函数 sl_WlanSmartConfigStart()扫描接入点，Wi-Fi 站点连接接入点，运行智能配置程序 SmartConfigCC3X，通过接入点对 CC3200 站点进行配置，配置成功后将配置文件保存在串行闪存中，同时建立与接入点的连接。

Provisioning_smartconfig 项目包含下列 4 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO 引脚配置程序文件。
- gpio_if.c：GPIO 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Provisioning_smartconfig 项目程序流程如图 7.2 所示。

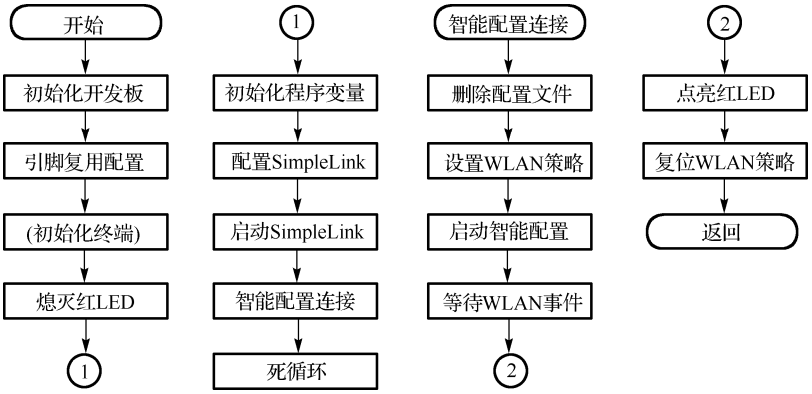


图 7.2　Provisioning_smartconfig 项目程序流程

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（2）在 CCS Project Explorer（项目浏览器）中单击 provisioning_smartconfig 项目，使其处于活动（Active）状态。

（3）单击主工具栏中的“Debug”（调试）按钮，生成输出文件 provisioning_smartconfig.out，并将 provisioning_smartconfig.out 下载到开发板的 CC3200 中，然后进入调试界面。

（4）单击调试工具栏中的“Resume”（运行）按钮运行程序，开发板上的红 LED 熄灭（默认已熄灭）。

（5）用手机连接 Wi-Fi 接入点，如 1234。

（6）在手机上运行智能配置程序 SmartConfigCC3X，应用程序自动显示搜索到的接入点名称（如 1234），在接入点名称下输入接入点的密码（Password），单击“Start”（开始）按钮，开始通过接入点对 CC3200 进行智能配置，“Start”（开始）按钮变为“Stop”（停止）。CC3200 智能配置成功后，开发板上的红 LED 点亮。

注意：如果开发板上的红 LED 长时间无法点亮，请参考 2.2.2 节中服务包更新，重新更新服务包。

（7）单击“Stop”（停止）按钮，停止智能配置。

（8）单击调试工具栏中的“Suspend”（暂停）按钮，停止程序的运行。

（9）单击调试工具栏中的“Restart”（重新开始）按钮，为程序的分析做准备。

下面主要介绍智能配置连接 SmartConfigConnect()的分析。智能配置连接 SmartConfigConnect()包括下列内容，其中最主要的是启动 WLAN 智能配置 sl_WlanSmartConfigStart()。

- 删除 WLAN 配置文件：sl_WlanProfileDel()。
- 设置 WLAN 策略：sl_WlanPolicySet()。
- 启动 WLAN 智能配置：sl_WlanSmartConfigStart()。

启动 WLAN 智能配置 sl_WlanSmartConfigStart()的调用格式如图 7.3 所示。

```
602    lRetVal = sl_WlanSmartConfigStart(0,                /*groupIdBitmask*/
603                                      SMART_CONFIG_CIPHER_NONE,    /*cipher*/
604                                      0,                /*publicKeyLen*/
605                                      0,                /*group1KeyLen*/
606                                      0,                /*group2KeyLen */
607                                      NULL,             /*publicKey */
608                                      NULL,             /*group1Key */
609                                      NULL);            /*group2Key*/
```

图 7.3　sl_WlanSmartConfigStart()调用格式

启动 WLAN 智能配置 sl_WlanSmartConfigStart()的函数格式为

```
_i16 sl_WlanSmartConfigStart(const _u32  groupIdBitmask,
                             const _u8   cipher,
                             const _u8   publicKeyLen,
                             const _u8   group1KeyLen,
                             const _u8   group2KeyLen,
                             const _u8*  pPublicKey,
                             const _u8*  pGroup1Key,
                             const _u8*  pGroup2Key);
```

功能：启动 WLAN 智能配置。

参数说明：

- groupIdBitmask：搜索组 ID 位屏蔽，范围是 BIT_0～BIT_15，每一位代表一个搜索组 ID，默认的搜索组 ID 是 BIT_0，除此之外还可以搜索两组。
- cipher：密码，传递值是 SMART_CONFIG_CIPHER_NONE（不检查密码）。cipher 包括下列值。
 - ✧ SMART_CONFIG_CIPHER_SFLASH：密码存放在串行闪存，0。
 - ✧ SMART_CONFIG_CIPHER_AES：AES 加密，1。
 - ✧ SMART_CONFIG_CIPHER_NONE：不检查密码，0xFF。
- publicKeyLen：公共密码长度，用于默认搜索组 ID（BIT_0），传递值是 0。
- group1KeyLen：搜索组 1 密码长度，传递值是 0。
- group2KeyLen：搜索组 2 密码长度，传递值是 0。
- pPublicKey：公共密码，用于默认搜索组 ID（BIT_0），传递值是 NULL（空）。
- pGroup1Key：搜索组 1 密码，传递值是 NULL（空）。
- pGroup2Key：搜索组 2 密码，传递值是 NULL（空）。

返回值：0 表示操作成功，接入点名称和密码自动以配置文件方式保存在串行闪存中。

sl_WlanSmartConfigStart()的声明在 wlan.h 中，代码在 wlan.c 中。

智能配置的目的是让 CC3200 获得接入点的名称和密码，方法是同时在手机上运行智能配置程序，通过连接的接入点无线发送接入点名称和密码。密码可以加密发送，加密方法通过 sl_WlanSmartConfigStart()的 cipher 等参数指定。

CC3200 通过搜索获得接入点名称和密码，并将搜索到的接入点名称和密码以配置文件方式保存在串行闪存中，同时用搜索到的接入点名称和密码建立与接入点的连接。

智能配置完成后，CC3200 产生下面的 WLAN 事件。

SL_WLAN_SMART_CONFIG_COMPLETE_EVENT：智能配置完成事件，3。

智能配置完成事件数据中包含接入点的名称（SSID，长度是 4），如图 7.4 所示。

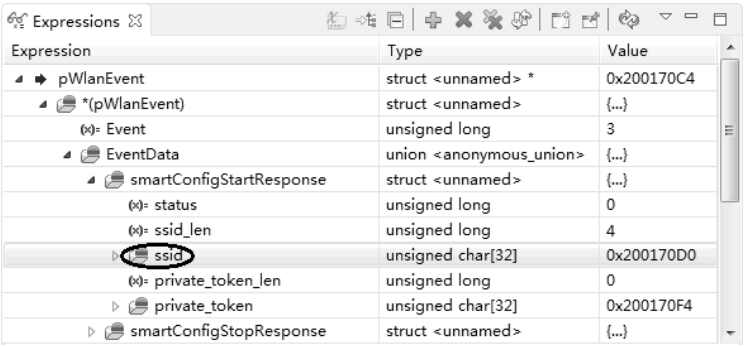


图 7.4　智能配置完成事件数据

WLAN 事件处理程序 SimpleLinkWlanEventHandler()对智能配置完成事件没做处理，只在超级终端中显示下列提示信息。

```
[WLAN EVENT] Unexpected event [0x3]          [WLAN 事件] 意外事件 [0x3]
```

建立与接入点的连接后，CC3200 产生下列 WLAN 事件和 NETAPP 事件。

- SL_WLAN_CONNECT_EVENT：WLAN 连接事件，1。
- SL_NETAPP_IPV4_IPACQUIRED_EVENT：获得 IPV4 事件，1。

WLAN 连接事件数据中包含接入点的名称（SSID）和 MAC 地址（BSSID），WLAN 事件处理程序 SimpleLinkWlanEventHandler()将设备状态 g_ulStatus 设置为 STATUS_BIT_CONNECTION（连接），并在超级终端中显示下列提示信息。

```
[WLAN EVENT] STA Connected to the AP: 1234 ,BSSID: 3c:e5:a6:1:7:bc
[WLAN 事件] 站点连接到接入点：1234，BSSID：3c:e5:a6:1:7:bc
```

获得 IPv4 事件数据中包含获得的 IP 地址和网关地址，NETAPP 事件处理程序 SimpleLinkNetAppEventHandler()将设备状态 g_ulStatus 设置为 STATUS_BIT_IP_AQUIRED（获得 IP），并在超级终端中显示下列提示信息。

```
[NETAPP EVENT] IP Acquired: IP=192.168.1.6 , Gateway=192.168.1.1
[NETAPP 事件] 获得 IP=192.168.1.6, 网关=192.168.1.1
```

设备状态 g_ulStatus 设置为 STATUS_BIT_CONNECTION（连接）和 STATUS_BIT_IP_AQUIRED（获得 IP）后，SmartConfigConnect()退出等待 WLAN 事件循环，点亮红 LED，复位 WLAN 策略，退出 SmartConfigConnect()。

注意：Provisioning_smartconfig 项目中定义了符号“NOTERM”，因此超级终端不工作。为了使超级终端工作，需要进行下列操作。

（1）在项目“属性”对话框中删除预定义符号“NOTERM”，如图 7.5 所示。

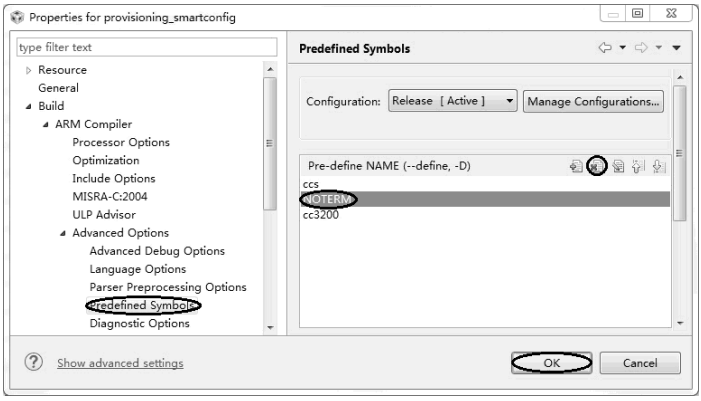


图 7.5　Provisioning_smartconfig 项目“属性”对话框

（2）在 pinmux.c 中添加下列 UART 引脚配置语句。

```
//Enable Peripheral Clocks
MAP_PRCMPeripheralClkEnable(PRCM_UARTA0, PRCM_RUN_MODE_CLK);
//Configure PIN_55 for UART0 UART0_TX
MAP_PinTypeUART(PIN_55, PIN_MODE_3);
//Configure PIN_57 for UART0 UART0_RX
MAP_PinTypeUART(PIN_57, PIN_MODE_3);
```

（3）在项目中添加 example\common 目录中的 uart_if.c 文件。

（4）在 PC 上打开超级终端。

（5）重新生成并运行程序，超级终端显示的内容如图 7.6 所示。

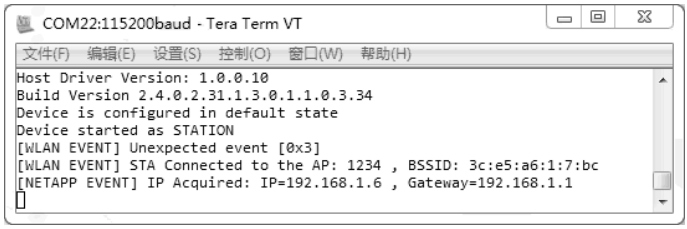


图 7.6　Provisioning_smartconfig 项目超级终端显示内容

sl_WlanSmartConfigStart()的应用参见 8.3 节。

7.2　Provisioning_wps 示例程序分析

Provisioning_wps 示例程序在“example\provisioning_wps”目录中，Provisioning_wps 项目的工作示意图如图 7.7 所示。



图 7.7　Provisioning_wps 项目工作示意图

CC3200 站点首先与接入点建立 WPS PBC 连接，按接入点上的“WPS”按钮或通过连接到接入点的 Wi-Fi 站点用浏览器进行 PBC 设置建立连接；然后断开连接，再与接入点建立 WPS PIN 连接，通过连接到接入点的 Wi-Fi 站点用浏览器进行 PIN 设置建立连接。

注意：使用 WPS 进行配置时，接入点必须支持“WPS 一键加密”（用手机或笔记本电脑扫描接入点后，接入点名称下方显示有“WPS 可用”）。

Provisioning_wps 项目包含下列 4 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO 引脚配置程序文件。
- gpio_if.c：GPIO 接口驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Provisioning_wps 项目程序流程如图 7.8 所示。

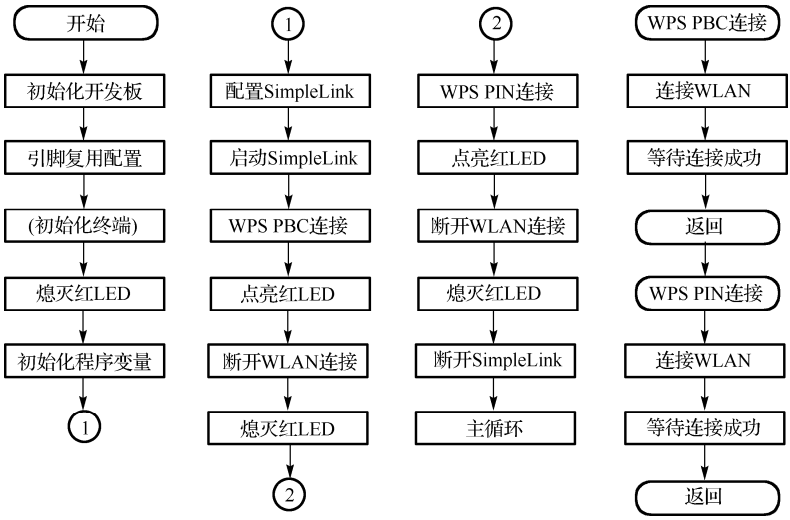


图 7.8 Provisioning_wps 项目程序流程

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（2）将 common.h 中的下列宏定义

```
#define SSID_NAME       "cc3200demo"        /* 接入点名称 */
```

修改为自己 AP 的配置值，例如：

```
#define SSID_NAME       "1234"              /* 接入点名称 */
```

（3）在 CCS Project Explorer（项目浏览器）中单击 provisioning_wps 项目，使其处于活动（Active）状态。

（4）单击主工具栏中的“Debug”（调试）按钮，生成输出文件 provisioning_wps. out，并将 provisioning_wps.out 下载到开发板的 CC3200 中，然后进入调试界面。

（5）单击调试工具栏中的“Resume”（运行）按钮运行程序，开发板上的红 LED 熄灭（默认已熄灭），等待 WPS PBC 连接。

（6）单击接入点上的“WPS”按钮或在连接到接入点的手机或笔记本电脑上用浏览器打开接入点主页“192.168.1.1”，单击 WPS 配置页面中 PBC 设置下的“开始连接”按钮进行连接，连接成功后开发板上的红 LED 点亮。

延时几秒后断开 WLAN 连接，开发板上的红 LED 熄灭，等待 WPS PIN 连接。

（7）在连接到接入点的手机或笔记本电脑上用浏览器打开接入点主页“192.168.1.1”，在 WPS 配置页面中 PIN 设置下输入 PIN“88664422”，单击“开始连接”按钮进行连接，连接成功后开发板上的红 LED 点亮。

延时几秒后断开 WLAN 连接，开发板上的红 LED 熄灭。

（8）单击调试工具栏中的“Suspend”（暂停）按钮，停止程序的运行。

（9）单击调试工具栏中的“Restart”（重新开始）按钮，为程序的分析做准备。

下面主要介绍 WPS PBC 连接 WpsConnectPushButton()和 WPS PIN 连接 WpsConnectPinCode()的分析。

7.2.1　WpsConnectPushButton()分析

WPS PBC 连接函数 WpsConnectPushButton()的内容如图 7.9 所示。

```
560 long
561 WpsConnectPushButton()
562 {
563     SlSecParams_t secParams;
564     long lRetVal = -1;
565
566     secParams.Key = "";
567     secParams.KeyLen = 0;
568     secParams.Type = SL_SEC_TYPE_WPS_PBC;
569
570     lRetVal = sl_WlanConnect(SSID_NAME, strlen(SSID_NAME), 0, &secParams,0);
571     ASSERT_ON_ERROR(lRetVal);
572
573     while((!IS_CONNECTED(g_ulStatus)) || (!IS_IP_ACQUIRED(g_ulStatus)))
574     {
575         _SlNonOsMainLoopTask();
576     }
577     return SUCCESS;
578 }
```

图 7.9　WpsConnectPushButton()函数内容

连接 WLAN sl_WlanConnect()的函数格式为：

```
_i16 sl_WlanConnect(_i8* pName, _i16 NameLen, _u8 *pMacAddr,
                    SlSecParams_t* pSecParams, SlSecParamsExt_t* pSecExtParams);
```

功能：连接 WLAN。

参数说明：

- pName：接入点名称指针，接入点名称 SSID_NAME 需要用户根据自己的实际情况在 common.h 中设置（如设置为 1234）。
- NameLen：接入点名称长度，传递值是 strlen(SSID_NAME)。
- pMacAddr：MAC 地址指针，传递值是 0。
- pSecParams：安全参数指针，安全参数传递变量名是 secParams，安全参数包括安全类型、安全密码和密码长度，WPS PBC 连接的安全类型是 SL_SEC_TYPE_WPS_PBC，表示 WPS PBC，3。WPS PBC 连接的安全密码为空，密码长度为 0。

- pSecExtParams：企业参数指针，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_WlanConnect()运行时调用 WLAN 事件处理程序 SimpleLinkWlanEventHandler()和 NETAPP 事件处理程序 SimpleLinkNetAppEventHandler()处理 WLAN 连接事件和 NETAPP 获得 IPV4 事件，分别将设备状态 g_ulStatus 设置为 STATUS_BIT_ CONNECTION（连接）和 STATUS_BIT_IP_ AQUIRED（获得 IP）。

7.2.2 WpsConnectPinCode()分析

WPS PIN 连接函数 WpsConnectPinCode()的内容如图 7.10 所示。

```
598 long
599 WpsConnectPinCode()
600 {
601     SlSecParams_t secParams;
602     long lRetVal = -1;
603
604     secParams.Key = WPS_PIN_CODE;
605     secParams.KeyLen = strlen(WPS_PIN_CODE);
606     secParams.Type = SL_SEC_TYPE_WPS_PIN;
607
608     lRetVal = sl_WlanConnect(SSID_NAME, strlen(SSID_NAME), 0, &secParams, 0);
609     ASSERT_ON_ERROR(lRetVal);
610
611     while((!IS_CONNECTED(g_ulStatus)) || (!IS_IP_ACQUIRED(g_ulStatus)))
612     {
613         _SlNonOsMainLoopTask();
614     }
615     return SUCCESS;
616 }
```

图 7.10 WpsConnectPinCode()函数内容

连接 WLAN sl_WlanConnect()的函数格式为

```
_i16 sl_WlanConnect(_i8* pName, _i16 NameLen, _u8 *pMacAddr,
              SlSecParams_t* pSecParams, SlSecParamsExt_t* pSecExtParams);
```

功能：连接 WLAN。

参数说明：

- pName：接入点名称指针，接入点名称 SSID_NAME 需要用户根据自己的实际情况在 common.h 中设置（如设置为 1234）。
- NameLen：接入点名称长度，传递值是 strlen(SSID_NAME)。
- pMacAddr：MAC 地址指针，传递值是 0。
- pSecParams：安全参数指针，安全参数传递变量名是 secParams，安全参数包括安全类型、安全密码和密码长度，WPS PIN 连接的安全类型是 SL_SEC_TYPE_WPS_PIN，表示 WPS PIN，4。WPS PBC 连接的安全密码为 WPS_PIN_CODE（88664422），密码长度为 strlen(WPS_ PIN_CODE)。
- pSecExtParams：企业参数指针，传递值是 0。

返回值：0 表示成功，负值表示错误。

sl_WlanConnect()运行时调用 WLAN 事件处理程序 SimpleLinkWlanEventHandler()和 NETAPP 事件处理程序 SimpleLinkNetAppEventHandler()处理 WLAN 连接事件和 NETAPP 获得 IPV4 事件，分别将设备状态 g_ulStatus 设置为 STATUS_BIT_ CONNECTION（连接）和 STATUS_BIT_IP_ AQUIRED（获得 IP）。

第 8 章

HTTP 服务器

本章在简单介绍 HTTP 服务器的基础上，首先介绍 Provisioning_ap 示例程序，然后介绍 Out_of_box 示例程序，特别是通过对 Out_of_box 示例程序的分析，加深对“2.2.1 节 TI CC3200 LaunchPad 开包即用体验”的理解和掌握。

8.1 HTTP 服务器简介

HTTP（Hypertext Transfer Protocol，超文本传输协议）是用于将超文本资源（HTML 网页、图像和查询结果等）传输到客户端的客户端/服务器协议，HTTP 工作在预定义的 TCP 套接字上，默认端口 80。

HTTP 服务器监听 HTTP 套接字（默认端口 80），按照请求类型（GET 或 POST）处理请求的 URI（Uniform Resource Identifier，统一资源标识符）资源和内容，并将 HTTP 响应返回给客户端。

HTTP 服务器可以与保存网页文件的串行闪存文件系统进行通信。保存在串行闪存中的文件有自己的文件名，文件名包含完整的路径。出于安全的目的，服务器访问文件系统的范围仅限于两个根文件夹“www/”和“www/safe/”。

HTTP 服务器在 ROM 中保存有一组作为内部默认网页的内部静态网页，这些内部网页可以提供开包即用的设备配置、设备状态和基本的分析工具。

CC3200 HTTP 服务器支持下列功能。

- HTTP 支持版本：1.0。
- HTTP 请求：GET 和 POST。
- 支持文件类型：.html、.htm、.css、.xml、.png 和.gif。
- HTML 表单使用 POST 方法提交数据（不支持 GET 方法表单提交）。
- HTTP 端口号可配置：默认端口 80。
- HTTP 服务器认证。
 - ✧ 可以允许或禁止（默认禁止）。
 - ✧ 认证名称、密码和域可配置。
- SimpleLink 域名（接入点模式）可配置。
- 内置默认网页提供设备配置、状态和分析工具。

- 为了安全，网页服务只能访问文件系统的下列根文件夹。
 - ✧ www/。
 - ✧ www/safe。
- 默认值如下。
 - ✧ 域名：www.mysimplelink.net 或 mysimplelink.net。
 - ✧ 认证名称：admin。
 - ✧ 认证密码：admin。
 - ✧ 认证域：Simple Link CC31xx。

8.1.1 HTTP GET 处理

当 HTTP 服务器获取 HTTP GET 请求时，首先检查请求的资源 URN（Uniform Resource Name，统一资源名称），然后检查请求的资源在串行闪存中是否存在，如果存在则将请求资源作为 HTTP 响应的一部分返回给客户端。

如果在串行闪存中找不到请求的资源，服务器检查请求的资源是否是 ROM 中的内部网页文件，如果是则返回，如果不是则发送 HTTP 错误信息“HTTP/1.0 404 Not Found（未找到）”。

1．默认网页

当 HTTP GET 不包含资源名称时，服务器按下列顺序查找文件。

- index.html。
- main.html。

服务器首先在串行闪存中查找上述文件，然后在内部网页中查找。

2．GET 标记

为了支持包含动态生成数据的网页，HTTP 服务器支持用一组预定义标记（Token）替换服务器动态生成的内容。

- 标记的长度固定为 10 个字符。
- 所有标记的前缀相同，长度是 7 个字符。
- 标记前缀是“__SL_G_”。
- 典型的标记格式是“__SL_G_XYZ”，其中“XYZ”是预定义标记。

3．用户定义标记

用户可以按照上述规则定义新的标记，如果服务器扫描 HTML 文件时发现非预定义的标记，则生成带标记名的“get_token_value”（GET 标记值）事件，从主机（MCU）请求标记值。

主机用带标记值的“send_token_value”（发送标记值）命令进行响应，服务器将标记值返回给客户端。如果主机没有响应“get_token_value”请求，服务器暂停 2 s 后用“Not Available”（不可用）作为标记值，并最终由浏览器显示。

注意：标记值的最大长度是 64 B。为了避免用户标记和内部标记冲突，用户标记使用“__SL_G_UXX”格式，其中“XX”可以包含任意字符或数字。

8.1.2　HTTP POST 处理

客户机使用 HTTP POST 请求更新 HTTP 服务器中的数据。HTTP 服务器支持内容类型为 application/x-www-form-urlencoded 的 HTML 表单，由浏览器发送的 POST 信息包含表单操作名和参数表，参数表包含一或多对变量名和变量值。

1．POST 标记

POST 中的变量名命名规则和 GET 标记相同。

- 标记的长度固定为 10 个字符。
- 所有标记的前缀相同，长度是 7 个字符。
- 标记前缀是“__SL_P_”。
- 典型的标记格式是“__SL_P_XYZ”，其中“XYZ”是用户定义标记。

当 HTTP 服务器接收到 POST 请求时，首先通过表单操作名检查 POST 是否需要内部处理，然后检查参数表，逐个检查参数表中的变量名是否与预定义标记匹配，如果匹配则对变量值进行处理。

2．POST 操作

POST 操作有两种类型：简单操作和复杂操作。对于简单 POST 操作，服务器不检查操作值，只对参数表进行处理，并保存新的信息（如设置域名）。对于复杂 POST 操作，服务器通过 POST 命令中的操作值对操作进行识别，并收集所有必要的 POST 参数，然后进行特定的操作（如添加配置文件）。

3．用户定义标记

用户定义标记用于在 POST 请求中向主机发送信息。如果服务器接收到的 POST 请求包含非预定义的标记，则生成“post_token_value”（POST 标记值）事件，向主机（MCU）发送表单操作名、标记名和标记值等信息，主机可以对这些信息进行处理。

注意：为了避免用户标记和内部标记冲突，用户标记使用“__SL_P_UXX”格式，其中“XX”可以包含任意字符或数字。

4．POST 后重定向

POST 处理后，HTTP 服务器检查已接收 POST 请求中的操作 URI，如果操作 URI 包含串行闪存或 ROM 中的有效网页，则服务器发送带操作 URI 值的响应“HTTP 302 Found（已找到）”完成重定向；如果不包含有效网页，则发送响应“HTTP 204 No content（无内容）”，浏览器停留在当前网页。

8.1.3 内部网页

CC3200 在 ROM 中嵌入了默认网页，可以用于完成下列工作。

- 获取设备的版本和一般信息。
- 配置 IP。
- 添加或删除 Wi-Fi 配置文件。
- 允许或禁止 Ping 测试。

内部网页的访问可以通过 API 配置，默认状态是允许访问。内部网页包括下列文件。

- about.html。
- image001.png。
- ip_config.html。
- logo.gif。
- main.html。
- ping.html。
- profiles_config.html。
- simple_link.css。
- status.html。
- tools.html

8.2 Provisioning_ap 示例程序分析

Provisioning_ap 示例程序在“example\provisioning_ap”目录中，Provisioning_ap 项目的工作示意图如图 8.1 所示。

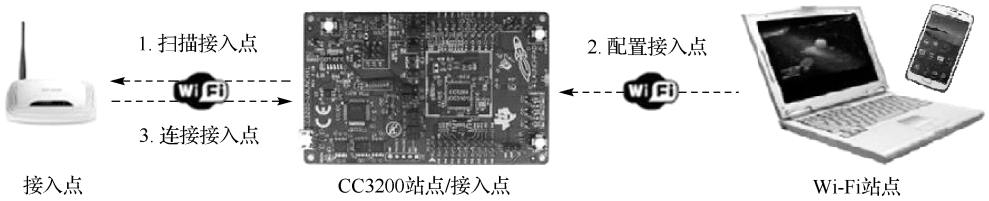


图 8.1 Provisioning_ap 项目工作示意图

CC3200 首先配置为站点模式，扫描并存储接入点信息；然后配置为接入点模式，当 Wi-Fi 站点浏览网页时，将扫描到的接入点名称通过 GET 标记发送到网页进行显示，供 Wi-Fi 站点进行选择配置，配置完成后通过 POST 标记将配置信息返回；最后配置为站点模式，利用配置信息连接接入点。

将 provisioning_ap 项目导入 CCS，项目包含下列 1 个程序文件和 3 个网页文件。

- main.c：主程序文件。
- main.html：主网页文件。

- param_status.html：设备状态参数网页文件。
- profile_config.html：配置文件设置网页文件。

Provisioning_ap 项目程序流程如图 8.2 所示。

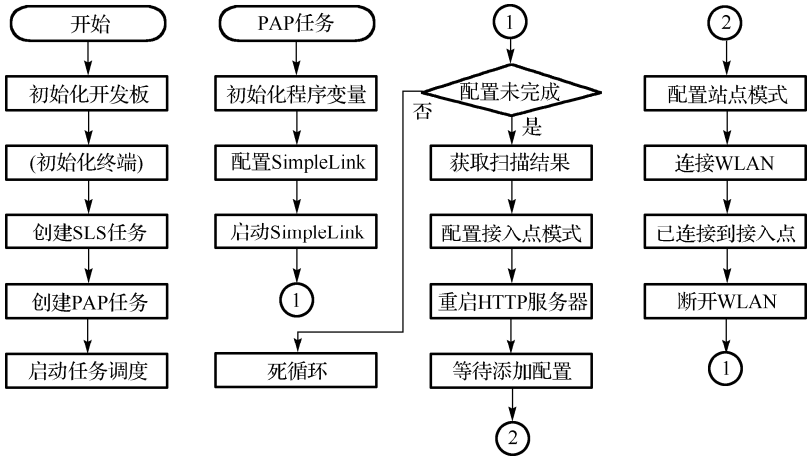


图 8.2　Provisioning_ap 项目程序流程图

在运行和分析 Provisioning_ap 项目程序前，需要用 UniFlash 将网页文件编程到串行闪存，具体步骤如下。

（1）将 CC3200 LaunchPad 上的 J15（1-2）用短路块连接。

（2）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（3）执行 CCS UniFlash - CC3xxx Edition 3.2.0（D:\TI\uniflash_3.2\eclipse\uniflash.exe），显示如图 8.3 所示的 CCS UniFlash 窗口。

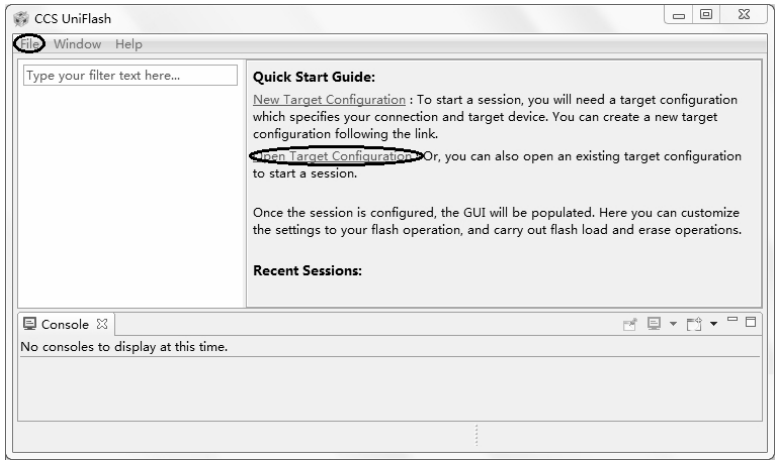


图 8.3　CCS UniFlash 窗口

（4）单击“File”（文件）菜单下的“Open Configuratiom”（打开配置）菜单项或单击“Quick Start Guide”（快速开始指南）下的“Open Target Configuratiom”（打开目标配置）打开如图 8.4 所示的“打开目标配置”对话框。



图 8.4　打开目标配置对话框

（5）单击“Browse”（浏览）按钮选择图 8.4 所示的目标配置文件，单击“OK”按钮显示如图 8.5 所示的目标配置窗口。

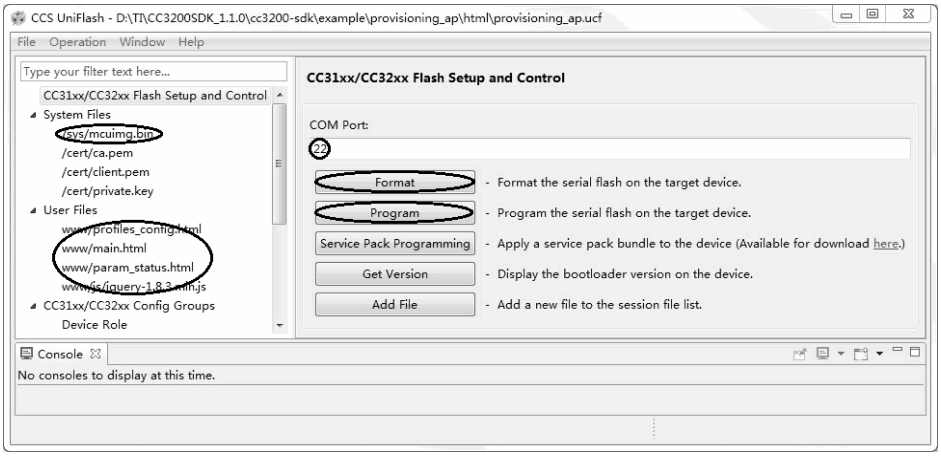


图 8.5　目标配置窗口

（6）在“COM Port”（COM 接口）下输入图 2.46 中显示的 UART 接口号“22”。

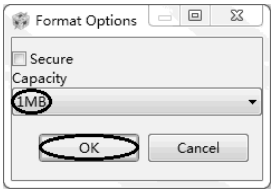


图 8.6　格式化选项对话框

（7）单击“Format”（格式化）按钮，打开“格式化选项”对话框，如图 8.6 所示。

（8）选择板载串行闪存的“Capacity”（容量），CC3200 LaunchPad 的板载串行闪存容量为 1 MB，单击“OK”按钮格式化串行闪存。

注意：格式化将删除串行闪存中的文件，一般情况下可以不进行格式化操作，编程时 UniFlash 根据设置自动更新同名文件。

如果 UniFlash 显示“Can't connect to device !!”（不能连接设备），请检查串口设置：如果 Tera Term 处于打开状态，请关闭 Tera Term。如果 Console（控制台）中显示“please restart the device”（请重新启动设备），请按开发板上的复位键重新启动设备。

（9）单击“Program”（编程）按钮，将应用程序二进制映像文件（mcuimg.bin）、网页文件（main. html 等）和 JavaScript 系统文件（jquery-1.8.3.min.js）编程到串行闪存中。

（10）编程完成后关闭 CCS UniFlash。

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）在 CCS Project Explorer（项目浏览器）中单击 provisioning_ap 项目，使其处于活动（Active）状态。

（2）单击主工具栏中的“Debug”（调试）按钮，生成输出文件 provisioning_ap.out，并将 provisioning_ap.out 下载到开发板的 CC3200 中，然后进入调试界面。

（3）单击调试工具栏中的“Resume”（运行）按钮运行程序。

（4）用手机或笔记本电脑连接 Wi-Fi 接入点“mysimplelink-054D6C”。

（5）在手机或笔记本电脑上打开浏览器，在地址栏中输入“mysimplelink.net”，打开串行闪存中的 main.html；单击 main.html 中的“Profiles”标签，打开 profiles_config.html，如图 8.7 所示。如果其中不包含用户接入点，可重新运行程序。

（6）选择要连接的接入点名称（如 1234）和安全类型（WPA），输入安全密码（1234asdf）和配置文件优先级（0），单击“Add”（添加）按钮，网页提示“Checking Status..”（检查状态），单击“确定”按钮开始配置接入点。

当网页提示“Successfully Connected to AP!!”（成功连接到 AP）时，单击“确定”按钮，网页 profiles_config.html 刷新，底部显示配置文件信息，如图 8.8 所示。

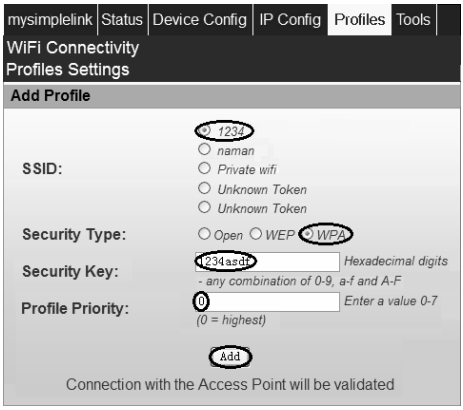


图 8.7　profiles_config.html 网页添加配置文件



图 8.8　profiles_config.html 网页配置文件信息

下面介绍程序的分析步骤，包括配置接入点任务（PAP）函数 ProvisioningAP()、获取扫描结果函数 GetScanResult()和 HTTP 服务器分析等，重点介绍 HTTP 服务器分析。

8.2.1　ProvisioningAP()分析

配置接入点任务（PAP）函数 ProvisioningAP()主要包括下列内容。

- 初始化程序变量：InitializeAppVariables()。
- 配置 SimpleLink 为默认状态：ConfigureSimpleLinkToDefaultState()。
- 启动 SimpleLink：sl_Start()。
- 获取扫描结果：GetScanResult()。
- 配置接入点模式：ConfigureMode()。
- 重启 HTTP 服务器：sl_NetAppStop() + sl_NetAppStart()。
- 配置站点模式：ConfigureMode()。
- 连接已配置接入点：WlanConnect()。
- 断开 WLAN 连接：sl_WlanDisconnect()。

1．初始化程序变量 InitializeAppVariables()

初始化程序变量函数 InitializeAppVariables()对下列全局变量进行初始化。

- g_ucProfileAdded：配置文件已添加，1。
- g_ucConnectedToConfAP：已连接到配置接入点，0。
- g_ucProvisioningDone：配置完成，0。
- g_ulGatewayIP：网关 IP 地址，0。
- g_ucConnectionSSID：接入点名称，0。
- g_ucConnectionBSSID：接入点 MAC 地址，0。

2．配置 SimpleLink 为默认状态 ConfigureSimpleLinkToDefaultState()

配置 SimpleLink 为默认状态函数通过调用 ConfigureSimpleLinkToDefaultState()实现。

3．启动 SimpleLink

启动 SimpleLink 通过调用 sl_Start()实现。

4．获取扫描结果 GetScanResult()

获取扫描结果 GetScanResult()将扫描到的接入点信息存放在全局数组 g_NetEntries[]中，详情参见 8.2.2 节。

5．配置接入点模式 ConfigureMode()

配置接入点模式通过调用 ConfigureMode()实现。

6．重启 HTTP 服务器

重启 HTTP 器服务通过停止 HTTP 服务器后再启动 HTTP 服务器实现。

（1）停止 HTTP 服务器的函数是 sl_NetAppStop()，函数格式为

```
_i16 sl_NetAppStop(_u32 AppBitMap);
```

功能：停止网络应用程序。

参数说明：AppBitMap，应用程序标识，传递值是 SL_NET_APP_HTTP_SERVER_ID，在 netapp.h 中定义为 1。AppBitMap 包括下列值。

- SL_NET_APP_HTTP_SERVER_ID：HTTP 服务器 ID，1。
- SL_NET_APP_DHCP_SERVER_ID：DHCP 服务器 ID，2。
- SL_NET_APP_MDNS_ID：MDNS ID，4。

返回值：0 表示成功，负值表示错误。

sl_NetAppStop()的声明在 netapp.h 中，代码在 netapp.c 中。

（2）启动 HTTP 服务器的函数是 sl_NetAppStart()，函数格式为

```
_i16 sl_NetAppStart(_u32 AppBitMap);
```

功能：启动网络应用程序。

参数说明：AppBitMap，应用程序标识，传递值是 SL_NET_APP_HTTP_SERVER_ID。

返回值：0 表示成功，负值表示错误。

sl_NetAppStart()的声明在 netapp.h 中，代码在 netapp.c 中。

重启 HTTP 服务器后，等待添加配置文件。用户通过 Wi-Fi 站点浏览设备中的网页，通过网页获取接入点参数：接入点名称、安全类型、安全密码和配置文件优先级（参见图 8.7）。获取接入点参数后，HTTP 服务器回调函数 SimpleLinkHttpServerCallback() 将 g_ucProfileAdded（配置文件已添加）设为 0，程序退出等待。

HTTP 服务器的详情参见 8.2.3 节。

7．配置站点模式 ConfigureMode()

配置站点模式通过调用 ConfigureMode()实现。

8．连接已配置接入点 WlanConnect()

连接已配置接入点通过调用 WlanConnect()实现，WlanConnect()首先调用添加配置文件函数 sl_WlanProfileAdd()将接入点参数添加到配置文件中，然后调用连接 WLAN 函数 sl_WlanConnect()利用接入点参数连接接入点。

添加配置文件 sl_WlanProfileAdd()的函数格式为

```
_i16 sl_WlanProfileAdd(_i8* pName, _i16 NameLen, _u8* pMacAddr,
              SlSecParams_t* pSecParams, SlSecParamsExt_t* pSecExtParams,
              _u32 Priority, _u32 Options);
```

功能：添加配置文件。

参数说明：

- pName：接入点名称指针。
- NameLen：接入点名称长度。
- pMacAddr：MAC 地址指针，传递值是 0。
- pSecParams：安全参数指针。
- pSecExtParams：企业参数指针，传递值是 0。
- Priority：配置文件优先级。
- Options：选项，传递值是 0。

返回值：正值表示成功，负值表示错误。

sl_WlanProfileAdd()的声明在 wlan.h 中，代码在 wlan.c 中。调用 WlanConnect()连接已配置接入点成功后，程序将 g_ucConnectedToConfAP（已连接到配置接入点）设为 1。

HTTP 服务器通过用户 GET 标记“__SL_G_US0”检查 g_ucConnectedToConfAP 的状态，连接已配置接入点成功后，在网页的 Profiles（配置文件）下显示接入点配置参数（参见图 8.8）。

9．断开 WLAN 连接 sl_WlanDisconnect()

断开 WLAN 连接通过调用 sl_WlanDisconnect()实现。

8.2.2 GetScanResult()分析

获取扫描结果函数 GetScanResult()主要包括下列内容。

- 设置 WLAN 策略：sl_WlanPolicySet()。
- 获得 WLAN 扫描结果：sl_WlanGetNetworkList()。

1．设置 WLAN 策略 sl_WlanPolicySet()

设置 WLAN 策略 sl_WlanPolicySet()的函数格式为

```
_i16 sl_WlanPolicySet(_u8 Type, const _u8 Policy, _u8 *pVal, _u8 ValLen);
```

功能：设置 WLAN 策略。

参数说明：

- Type：策略类型，传递值分别是 SL_POLICY_CONNECTION 和 SL_POLICY_SCAN。Type 包括下列值。
 - ✧ SL_POLICY_CONNECTION：连接策略，0x10。
 - ✧ SL_POLICY_SCAN：扫描策略，0x20。
 - ✧ SL_POLICY_PM：电源管理策略，0x30。
 - ✧ SL_POLICY_P2P：点对点策略，0x40。
- Policy：策略值，不同策略类型的策略值不同。连接策略通过“SL_CONNECTION_POLICY(Auto, Fast, Open, anyP2P, autoSmartConfig)”指定，Auto=1 表示自动连接，Fast=1 表示快速连接，anyP2P=1 表示点对点连接，autoSmartConfig=1 表示自动智能配置连接。扫描策略通过“SL_SCAN_POLICY(Enable)”指定，Enable=1 表示允许扫描，Enable=0 表示禁止扫描。允许扫描时通过选项值指定扫描时间间隔，默认值是 10 min，指定值是 1 min（IntervalVal = 60）。
- pVal：选项值指针，连接策略的选项值为 NULL，扫描策略的选项值为扫描时间间隔。
- ValLen：选项值长度。

返回值：0 表示成功，负值表示错误。

sl_WlanPolicySet()的声明在 wlan.h 中，代码在 wlan.c 中。

2．获得 WLAN 扫描结果 sl_WlanGetNetworkList()

获得 WLAN 扫描结果 sl_WlanGetNetworkList()的函数格式为

```
_i16 sl_WlanGetNetworkList(_u8 Index, _u8 Count,
                           Sl_WlanNetworkEntry_t *pEntries);
```

功能：获得 WLAN 扫描结果。

参数说明：

- Index：获取扫描结果起始索引标识（0～19），传递值是 0。
- Count：获取数量，最大值是 20–Index，传递值是 SCAN_TABLE_SIZE（20）。
- pEntries：结果指针，指向存放结果的全局数组 g_NetEntries[]，其中包含 ssid（接入

点名称）、ssid_len（接入点名称长度）、sec_type（安全类型）、bssid（接入点 MAC 地址）和 rssi（接收信号强度）等。

返回值：实际获取数量。

sl_WlanGetNetworkList()的声明在 wlan.h 中，代码在 wlan.c 中。

8.2.3　HTTP 服务器分析

HTTP 服务器分析包括回调函数分析和网页分析。

1．回调函数分析

HTTP 服务器回调函数 SimpleLinkHttpServerCallback()包括 GET 标记值事件处理和 POST 标记值事件处理两部分，事件处理流程如图 8.9 所示。

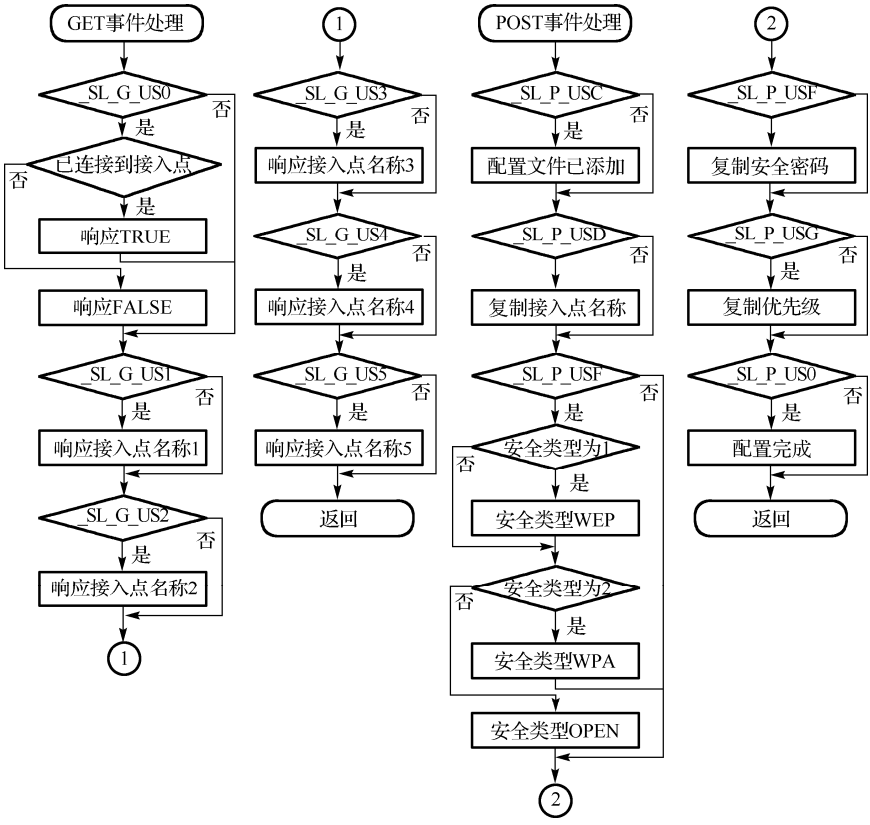


图 8.9　Provisioning_ap 项目回调函数事件处理流程

字符串数组 g_token_get[]中定义了 6 个 GET 标记“__SL_G_US0～5”，其中“__SL_G_US0”用于检查 g_ucConnectedToConfAP 的状态（通过 Java 函数$.ajax()实现），“__SL_G_US1～5”用于从 g_NetEntries[]中获取 5 个接入点的名称。

注意：因为“__SL_G_US0”返回的数据需要处理，而“__SL_G_US1～5”返回的数据只需显示，所以“__SL_G_US0”需要通过 Java 脚本实现，而“__SL_G_US1～5”只需在网页中通过用户 GET 标记即可实现。

网页中定义了 6 个 POST 标记“__SL_P_USC～G” 和“__SL_P_US0”，其中“__SL_P_USC”用于发送标记值“Add”，“__SL_P_USD～G”用于发送接入点名称、安全类型、安全密码和配置文件优先级，“__SL_P_US0”用于通知设备配置完成（未实现）。

HTTP 服务器回调函数 SimpleLinkHttpServerCallback()的格式为

```
void SimpleLinkHttpServerCallback(
                          SlHttpServerEvent_t *pSlHttpServerEvent,
                          SlHttpServerResponse_t *pSlHttpServerResponse);
```

功能：HTTP 服务器回调处理。

参数说明：

- pSlHttpServerEvent：HTTP 服务器事件指针，其中包括事件号和事件数据。
- pSlHttpServerResponse：HTTP 服务器响应数据指针，响应数据即 GET 数据。

Provisioning_ap 项目 GET 标记值事件数据如图 8.10 所示。

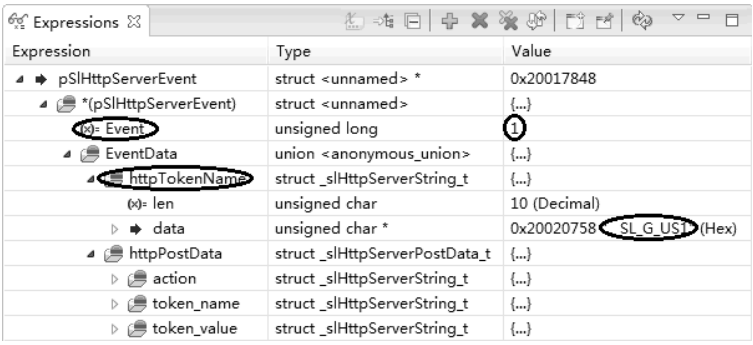


图 8.10　Provisioning_ap 项目 GET 标记值事件数据

GET 标记值事件数据中 Event（事件）的值为 SL_NETAPP_HTTPGETTOKENVALUE_EVENT：GET 标记值事件，1。

GET 标记值事件数据 httpTokenName 中包含标记名“__SL_G_US1”。

Provisioning_ap 项目 GET 标记值响应数据如图 8.11 所示。

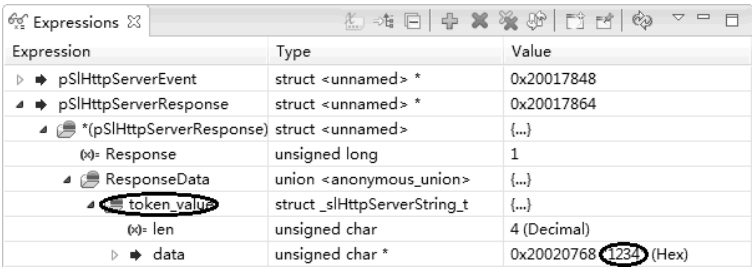


图 8.11　Provisioning_ap 项目 GET 标记值响应数据

响应数据 token_value 中包含返回的标记值“1234”（接入点名称 1）。

Provisioning_ap 项目 POST 标记值事件数据如图 8.12 所示。

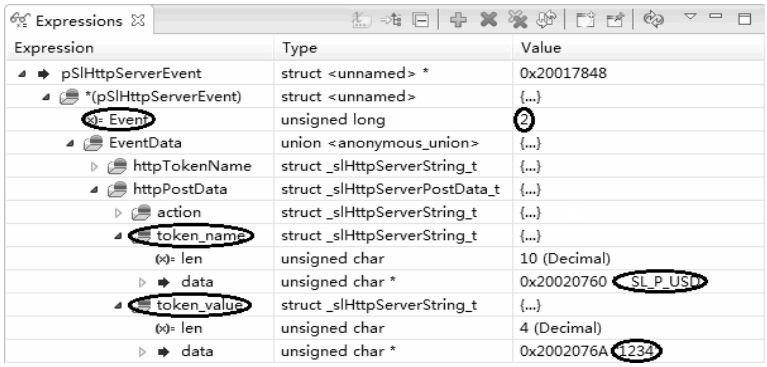


图 8.12　Provisioning_ap 项目 POST 标记值事件数据

POST 标记值事件数据中 Event（事件）的值为 SL_NETAPP_HTTPPOSTTOKENVALUE_EVENT：POST 标记值事件，2。

POST 标记值事件数据 token_name 中包含标记名“__SL_P_USD”，token_value 中包含标记值“1234”（已配置接入点名称）。

2．网页分析

网页分析包括表单分析和 Java 脚本分析。

（1）表单分析。网页 profile_config.html 中“Add Profile”（添加配置文件）表单对应的 HTML 代码如图 8.13 所示。

```
<form method="POST" name="SimpleLink Configuration" action = "nothing" onsubmit="on_submit(this)">
<tr><td class = "empty-line" colspan=2></td></tr>
<tr><td class = "frame-title" colspan=2> Add Profile </td></tr>
<tr><td class = "empty-l-in-box" colspan=2></td></tr>

<tr><td class = "label" dir=LTR> SSID: </td>
    <td class = "user-input" dir=LTR>
        <input type = "radio" name = "__SL_P_USD" value = "__SL_G_US1" > __SL_G_US1 </input><br>
        <input type = "radio" name = "__SL_P_USD" value = "__SL_G_US2" > __SL_G_US2 </input><br>
        <input type = "radio" name = "__SL_P_USD" value = "__SL_G_US3" > __SL_G_US3 </input><br>
        <input type = "radio" name = "__SL_P_USD" value = "__SL_G_US4" > __SL_G_US4 </input><br>
        <input type = "radio" name = "__SL_P_USD" value = "__SL_G_US5" > __SL_G_US5 </input></td></tr>

<tr><td class = "label"> Security Type: </td>
    <td class = "user-input">
        <input type = "radio" name = "__SL_P_USE" value = "0" checked />Open
        <input type = "radio" name = "__SL_P_USE" value = "1" />WEP
        <input type = "radio" name = "__SL_P_USE" value = "2" />WPA</tr>

<tr><td class = "label"> Security Key: </td>
    <td class = "user-input">
        <input type = "text" maxlength = "64" name = "__SL_P_USP" /> Hexadecimal digits - any combination of 0-9

<tr><td class = "label"> Profile Priority: </td>
    <td class = "user-input">
        <input type = "text" maxlength = "1" name = "__SL_P_USG" /> Enter a value 0-7 (0 = highest)</td></tr>

<tr><td class = "empty-l-in-box" colspan=2></td></tr>
<tr><td class = "in-a-box" colspan=2 align=center>
        <input type = "submit" name = "__SL_P_USL" value="Add" /></td></tr>

<tr><td class = "in-a-box" colspan=2 align=center> Connection with the Access Point will be validated</td></tr>
<tr><td class = "empty-l-in-box" colspan=2></td></tr>
<tr><td class = "border-l-top" colspan=2></td></tr>
</form>
```

图 8.13　网页 profile_config.html 中“添加配置文件”表单对应的 HTML 代码

选择的接入点名称（SSID）通过 POST 标记“__SL_P_USD”发送给主机，接入点名称通过 GET 标记“__SL_G_US1～5”从主机获取，安全类型（Security Type）、安全密码（Security Key）和配置文件优先级（Profile Priority）分别通过 POST 标记“__SL_P_USE～G”发送给主机，提交（submit）按钮通过 POST 标记“__SL_P_USC”将“Add”发送给主机。

（2）Java 脚本分析。Java 脚本主要实现“连接到接入点”状态的获取和处理，获取标记值 getTokenValue()函数的 Java 定义如图 8.14 所示。

```
// helper function for getting token values
function getTokenValue(paramPage, tokenId, successFn, failFn) {
    $.ajax({
        "type": "GET",
        "url": paramPage,
        "cache": false,
        "dataType": "html"
    })
    // define callback for get request success
    .done(function(getdata, status, xhr) {
        successFn($(getdata).filter('#'+tokenId).text());
    })
    .fail(function(jqxhr, settings, exception) {
        failFn();
    });
}
```

图 8.14　网页 profile_config.html 中 getTokenValue()函数的 Java 定义

获取标记值函数 getTokenValue()包含下列 4 个参数。

- paramPage：参数网页，其中包含 GET 标记。
- tokenId：标记标识。
- successFn：成功函数。
- failFn：失败函数。

获取标记值函数 getTokenValue()的核心是$.ajax()函数，$.ajax()通过 HTTP 请求加载远程数据，$.ajax()包含下列 4 个参数。

- type：指定 HTTP 请求类型，GET 或 POST，默认为 GET。
- url：指定 HTTP 请求地址。
- cache：指定是否从缓存中加载数据，true 或 false，默认为 true。
- dataType：指定 HTTP 请求数据类型，html 为纯文本 HTML 信息。

$.ajax()的回调函数 done()包含下列 3 个参数。

- getdata：指定 HTTP 请求返回的数据。
- status：指定 HTTP 请求状态，success（成功）。
- xhr：XMLHttpRequest 对象。

回调函数 done()的内容是成功函数 successFn()，successFn()从返回数据 getdata 中提取出 tokenId 对应的标记值。

调用获取标记值函数 getTokenValue()的 4 个参数如下。

- paramPage：传递值是 param_status.html，其中包含 GET 标记“__SL_G_US0”和标记标识“Device_Status”（设备状态）。

- tokenId：传递值是“Device_Status”（设备状态）。
- successFn：根据 GET 标记“__SL_G_US0”的返回值进行相应的处理，在弹出对话框中显示提示信息，调用 location.reload()刷新页面。
- failFn：在弹出对话框中显示提示信息。

获取标记值函数 getTokenValue()包含在获取状态函数 get_status()中，获取状态函数 get_status()通过设置间隔函数 setInterval()设置为每 5 000 ms（5 s）调用一次。

初始情况下，标志变量“_flag = 0”，获取状态函数 get_status()不调用获取标记值函数 getTokenValue()；当单击“Add Profile”（添加配置文件）表单中的“Add”按钮时，调用 on_submit()函数，将_flag 设置为 1，获取状态函数 get_status()将_flag 复位为 0，同时调用获取标记值函数 getTokenValue()通过 GET 标记“__SL_G_US0”获取“连接到接入点”的状态，并调用成功函数进行相应的处理。

注意：用笔记本电脑无线连接 CC33200 浏览网页时，可以按“F12”键打开“开发人员工具”窗口对 Java 脚本进行调试，具体方法和 CCS 调试类似。

8.3　Out_of_box 示例程序分析

Out_of_box 示例程序在“example\out_of_box”目录中，Out_of_box 项目的工作示意图如图 8.15 所示。

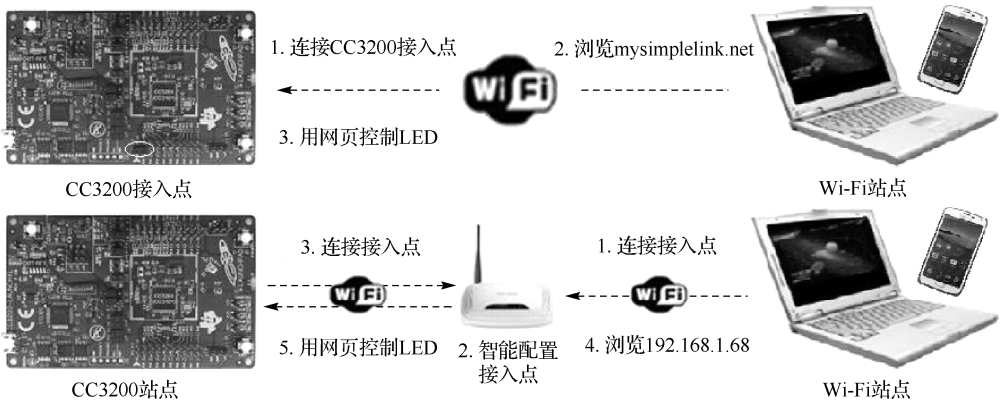


图 8.15　Out_of_box 项目工作示意图

当 CC3200 配置为接入点时，Wi-Fi 站点首先连接 CC3200 接入点，然后浏览 CC3200 的默认主页 mysimplelink.net，并用网页与 CC3200 通信。

当 CC3200 配置为站点时，Wi-Fi 站点首先连接接入点，并通过接入点对 CC3200 站点进行智能配置（SmartConfig），配置成功后 CC3200 站点连接接入点，然后 Wi-Fi 站点通过接入点使用 IP 地址（如 192.168.1.6）浏览 CC3200 站点，并用网页与 CC3200 通信。

Out_of_box 项目包含下列 10 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO、UART 和 I2C 引脚配置程序文件。

- gpio_if.c：GPIO 接口驱动程序文件。
- uart_if.c：UART 接口驱动程序文件。
- i2c_if.c：I2C 接口驱动程序文件。
- tmp006drv.c：温度传感器 TMP006 驱动程序文件。
- bma222drv.c：加速度传感器 BMA222 驱动程序文件。
- device_status.c：设备状态程序文件。
- smartconfig.c：智能配置程序文件。
- startup_ccs.c：CCS 启动程序文件。

还包含 14 个网页文件、17 个图像文件和 6 个 Java 脚本文件，其中主要文件如下。

- index.html：主网页文件。
- demos.html：演示网页文件。
- demos-sprinkler.html：喷水演示网页文件。
- demos-appliances.html：家电演示网页文件。
- demos-security.html：安全演示网页文件。
- demos-energy.html：能源演示网页文件。
- param_demos.html：演示参数网页文件。
- page-actions.js：网页操作脚本文件。
- com.TI.tokenPoller.js：TI 标记轮询脚本文件。
- jquery-1.8.3.min.js：Java 脚本系统文件。

Out_of_box 项目程序流程如图 8.16 所示。

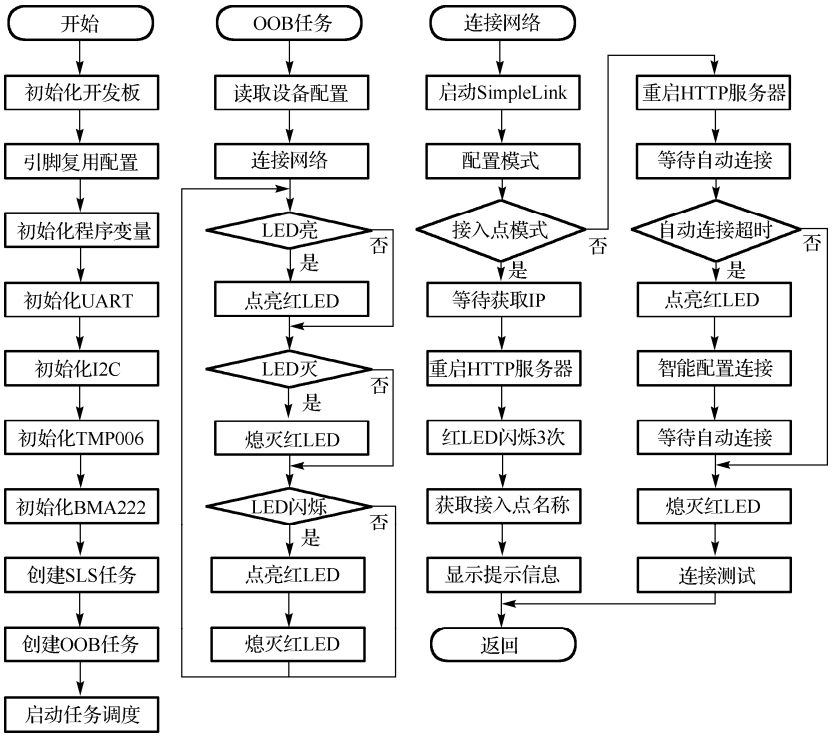


图 8.16　Out_of_box 项目程序流程

在运行和分析 Out_of_box 项目程序前，需要用 UniFlash 将网页文件、图像文件和脚本文件编程到串行闪存，具体步骤可参考 Provisioning_ap 项目的操作，不同之处是图 8.4 中打开的目标配置文件为“D:\TI\CC3200SDK_1.1.0\cc3200-sdk\example\out_of_box\html\out_of_box.ucf”。

分析程序前首先运行程序，观察程序结果，步骤如下。

（1）拔掉 CC3200 LaunchPad 上连接 J15（1-2）的短路块，将 CC3200 LaunchPad 上的 J22.P1（1-2）用短路块连接。

警告：用短路块连接 J22.P1（1-2）时要特别注意，不要误将短路块连接到 J22.P3（1-2）上，否则会造成电源短路（J22.P3.1 是 5 V 电源，J22.P3.2 是 GND）！

为了防止误连接，可以将 J22.P3.1 往外掰弯。

（2）将 CC3200 LaunchPad 通过 USB 线与 PC 连接。

（3）在 PC 上打开超级终端。

（4）按一下 CC3200 LaunchPad 上的“RESET”（复位）按键，运行串行闪存中的程序，开发板上的红 LED 闪烁 3 次，超级终端中显示下列信息。

```
*******************************************************************************
                    CC3200 Out of Box Application
*******************************************************************************

Manufacturer ID: 0x5449                         TMP006 制造商标识：0x5449
Device ID: 0x67                                 TMP006 器件标识：0x67
Configuration register value: 0x7480            TMP006 配置寄存器值：0x7480
CHIP ID: 0xf8                                   BMA222 芯片标识：0xf8
[NETAPP EVENT] IP Acquired: IP=192.168.1.1 , Gateway=192.168.1.1

Connect to : 'mysimplelink-054D6C'     连接到接入点“mysimplelink-054D6C”
```

（5）参考 2.2.1 节的（2）～（12）完成应用演示操作。

（6）拔掉 CC3200 LaunchPad 上连接 J22.P1（1-2）的短路块。

（7）再按一下 CC3200 LaunchPad 上的“RESET”（复位）按键，重新运行程序，开发板上的红 LED 快速闪烁，然后点亮，超级终端中显示下列信息。

```
*******************************************************************************
                    CC3200 Out of Box Application
*******************************************************************************

Manufacturer ID: 0x5449                         TMP006 制造商标识：0x5449
Device ID: 0x67                                 TMP006 器件标识：0x67
Configuration register value: 0x7480            TMP006 配置寄存器值：0x7480
CHIP ID: 0xf8                                   BMA222 芯片标识：0xf8
```

（8）在手机中打开 SmartConfigCC3X 应用程序，应用程序自动显示搜索到的接入点名称（如 1234），在接入点名称下输入接入点的密码（Password），单击“Start”（开始）按钮，开始通过接入点对 CC3200 进行智能配置。

CC3200 智能配置成功后，开发板上的红 LED 熄灭，超级终端中显示下列信息。

```
[WLAN EVENT] Devive Connected to the AP: 1234 , BSSID: 3c:e5:a6:1:7:bc
[WLAN 事件] 设备连接到接入点：1234，BSSID：3c:e5:a6:1:7:bc
[NETAPP EVENT] IP Acquired: IP=192.168.1.6 , Gateway=192.168.1.1
[NETAPP 事件] 获得 IP=192.168.1.6，网关=192.168.1.1
```

注意：Out_of_box 项目没有进行“配置 SimpleLink 为默认状态”操作，因此如果串行闪存中存有配置文件，则不需要智能配置，设备利用配置文件自动连接接入点，并在超级终端中显示上述信息。

（9）用手机或笔记本电脑连接 Wi-Fi 接入点“1234”，打开浏览器，在地址栏中输入“192.168.1.6”，打开开发板上串行闪存中的网页。

（10）参考 2.2.1 节（4）～（12）完成应用演示操作。

和设备配置为接入点模式不同，设备配置为站点时，设备会进行连接测试，测试是否能 Ping 通“www.ti.com”。如果 Ping 通，则图 2.47 上方的 Current Status（当前状态）会由“Offline”（离线）变为“Online”（在线）。

下面主要介绍 HTTP 服务器回调函数分析以及家庭自动化（Home Automation）、家用电器（Home Appliances）、安全保障（Safety and Security）和智能能源（Smart Energy）4 个应用演示功能的网页分析。

8.3.1 HTTP 服务器回调函数分析

Out_of_box 项目 HTTP 服务器回调函数 SimpleLinkHttpServerCallback()包括 GET 标记值事件处理和 POST 标记值事件处理两部分，事件处理流程如图 8.17 所示。

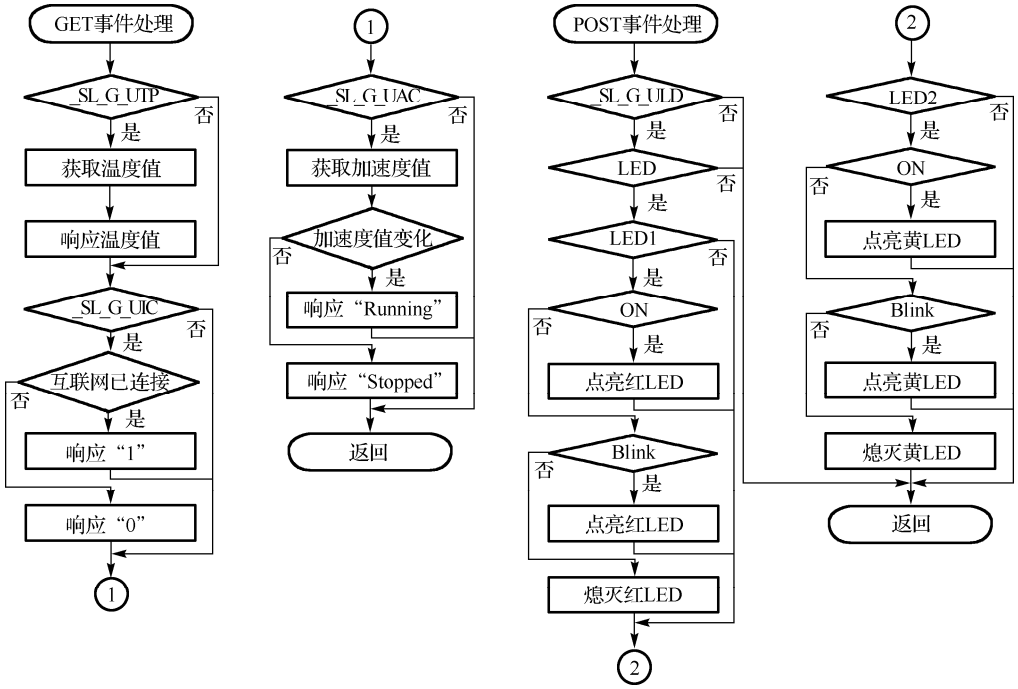


图 8.17 Out_of_box 项目回调函数事件处理流程

主程序中定义了 3 个 GET 标记：“__SL_G_UTP”用于获取温度值，“__SL_G_UAC”用于获取加速度值，并根据加速度值是否变化响应“Running”（运行）或“Stopped”（停止），“__SL_G_UIC”用于获取互联网连接状态。

获取温度值包括下列内容。

（1）比较（memcmp）EventData（事件数据）httpTokenName（HTTP 标记名）data（数据）是否为 GET_token_TEMP（__SL_G_UTP）。

（2）如果是则首先调用 TMP006DrvGetTemp()获得温度数据（cTemp=78），然后调用 itoa()将温度数据转换成字符串（"78"），并在温度字符串后添加空格（' '）和单位（'F'：华氏度），最后把字符串长度（sTempLen=3，应为 4）赋值给 ResponseData（响应数据）token_value（标记值）len（长度）。

注意：语句“ptr[sTempLen] = 'F';”应为“ptr[sTempLen++] = 'F';”。

获取加速度值包括下列内容。

（1）比较（memcmp）EventData（事件数据）httpTokenName（HTTP 标记名）data（数据）是否为 GET_token_ACC（__SL_G_UAC）。

（2）如果是则调用 ReadAccSensor()判断加速度数据是否变化。

（3）如果变化则把“Running”赋值给 ResponseData（响应数据）token_value（标记值）data（数据），把“Running”的长度赋值给 ResponseData（响应数据）token_value（标记值）len（长度）。

（4）否则把“Stopped”赋值给 ResponseData（响应数据）token_value（标记值）data（数据），把“Stopped”的长度赋值给 ResponseData（响应数据）token_value（标记值）len（长度）。

主程序还定义了 1 个 POST 标记“__SL_P_ULD”，用于控制两个 LED 的点亮（ON）、闪烁（Blink）或熄灭（OFF）。

POST 事件处理包括下列内容。

（1）比较（memcmp）EventData（事件数据）httpPostData（HTTP POST 数据）token_name（标记名）data（数据）是否为 POST_token（__SL_P_ULD）。

（2）如果是则比较 EventData（事件数据）httpPostData（HTTP POST 数据）token_value（标记值）data（数据）的前 3 个字符是否为“LED”，LED 是否是“1”，是“ON”或是“Blink”。

（3）如果是“LED1_ON”则调用 GPIO_IF_LedOn()点亮红 LED，同时将 LED 状态 g_ucLEDStatus 设为“LED_ON”，OOB 任务函数 OOBTask()中通过判断 g_ucLEDStatus 再次调用 GPIO_IF_ LedOn()点亮红 LED。

（4）如果是“LED1_Blink”则调用 GPIO_IF_LedOn()点亮红 LED，同时将 LED 状态 g_ucLEDStatus 设为“LED_BLINK”，OOB 任务函数 OOBTask()中通过判断 g_ucLEDStatus 调用 GPIO_IF_LedOn()和 GPIO_IF_LedOff()实现红 LED 闪烁。

（5）否则调用 GPIO_IF_LedOff()熄灭红 LED，同时将 LED 状态 g_ucLEDStatus 设为“LED_OFF”，OOB 任务函数 OOBTask()中通过判断 g_ucLEDStatus 再次调用 GPIO_IF_ LedOff()熄灭红 LED。

注意：虽然 POST 事件处理程序中有 LED2（黄 LED）的处理程序，但由于连接 LED2 的引脚 PIN_01 被 I2C 占用，所以 Out_of_box 项目中 LED2 无法使用。

8.3.2 家庭自动化（Home Automation）网页分析

家庭自动化（Home Automation）演示网页 demos-sprinkler.html 中喷水器开关和图像显示对应的 HTML 代码如图 8.18 所示。

```
<div class="switchControl" id="sprinklerSwitch">
    <div class="sprinklerOn">On</div>
    <div id="switchImage"></div>
    <div class="sprinklerOff">Off</div>
</div>
<img src="images/demo-sprinkler-off.jpg" id="sprinkler"
```

图 8.18 网页 demos-sprinkler.html 中喷水器开关和图像显示对应 HTML 代码

当单击网页上的“sprinklerSwitch”（喷水器开关）时，调用 sprinklerSwitch.click()函数，sprinklerSwitch.click()函数的内容如图 8.19 所示。

```
sprinklerSwitch.click(function(){
    if (switchBtn.hasClass('on')) {
        switchBtn.removeClass('on');
        com.TI.toggleLED('_OFF');
        spriklerRunning = false;
        sprinkler.attr('src',"images/demo-sprinkler-off.jpg");
        imageOn = false;
    } else {
        switchBtn.addClass('on');
        com.TI.toggleLED('_Blink');
        spriklerRunning = true;
        imageOn = true;
    }
});
```

图 8.19 网页 demos-sprinkler.html 中 sprinklerSwitch.click()函数

函数 sprinklerSwitch.click()首先判断 switchBtn 是否有“on”类（喷水器开关状态），如果有说明喷水器开关原来是“ON”，则进行“OFF”处理，否则进行“Blink”处理。

“OFF”处理：删除 switchBtn“on”类，发送 HTTP 请求“_OFF”，喷水器运行状态（spriklerRunning）和图像状态（imageOn）设为“false”，喷水器图像显示为“images/demo-sprinkler-off.jpg”。

注意：喷水器开关的显示切换在 style.css 中通过#switchImage{}和#switchImage.on{}实现，“关”时显示开关图像的下半部分，“开”时显示开关图像的下半部分。

“Blink”处理：添加 switchBtn“on”类，发送 HTTP 请求“_Blink”，喷水器运行状态（spriklerRunning）和图像状态（imageOn）设为“true”，喷水器图像通过调用 setInterval()每隔 300ms 分别显示“images/demo-sprinkler-off.jpg”和“images/demo-sprinkler-on.jpg”。

发送 HTTP 请求通过调用 com.TI.toggleLED()函数实现，com.TI.toggleLED()函数代码在 js/page-actions.js 文件中，内容如图 8.20 所示。

函数 com.TI.toggleLED()首先定义变量 params：

- “OFF”时 params = "LED1_OFF"。
- “Blink”时 params = "LED1_Blink"。

```
if(!window.com) com = new Object();
if(!com.TI) com.TI = new Object();
com.TI.toggleLED = function(whichOne) {
    var LEDnum = "1",
        params = "LED"+LEDnum;
    params += whichOne;
    $.post("No_content", {"__SL_P_ULD": params});
}
```

图 8.20　文件 page-actions.js 中 com.TI.toggleLED()函数

然后处理 HTTP 请求：以“post”方法发送“__SL_P_ULD”和 params。

HTTP 服务器接收 HTTP 请求后，对用户 POST 标记“__SL_P_ULD”调用回调函数 POST 事件处理程序对 params 参数传递的内容进行处理。

（1）对 LED1_OFF，调用 GPIO_IF_LedOff()熄灭红 LED，同时将 LED 状态 g_ucLEDStatus 设为 LED_OFF，OOB 任务函数 OOBTask()通过判断 g_ucLEDStatus 再次调用 GPIO_IF_LedOff()熄灭红 LED。

（2）对 LED1_Blink，调用 GPIO_IF_LedOn()点亮红 LED，同时将 LED 状态 g_ucLEDStatus 设为 LED_BLINK，OOB 任务函数 OOBTask()通过判断 g_ucLEDStatus 调用 GPIO_IF_LedOn()和 GPIO_IF_LedOff()实现红 LED 闪烁。

8.3.3　家用电器（Home Appliances）网页分析

家用电器（Home Appliances）演示功能和家庭自动化（Home Automation）演示功能不同，家庭自动化（Home Automation）是用 POST 方法控制开发板上的 LED，家用电器（Home Appliances）则是用 GET 方法将开发板上温度传感器和加速度传感器的结果显示在网页上。

网页 demos-appliances.html 中洗衣机和烘干机显示对应的 HTML 代码如图 8.21 所示。

```
<div class="washDryer">
    <div class="washerText">Water Temp:<br /><span id="Device_Temperature_out"><i>reloading</i></span></div>
    <div class="dryerText">Dryer:<br /><span id="Device_Accelerometer_out"><i>reloading</i></span></div>
    <br style="clear:both;" /><br>
    <img src="images/demo-washerDryer.jpg" name="washerDryerImage" width="300" height="210" id="washerDryerImage" />
</div>
```

图 8.21　网页 demos-appliances.html 中洗衣机和烘干机显示对应 HTML 代码

其中，水温（Water Temp，id=”Device_Temerature_out”）用于显示开发板上温度传感器测量的温度值，烘干机（Dryer，id=”Device_Accelerometer_out”）用于显示开发板上加速度传感器测量的数值变化情况（烘干机状态）。

为了实现温度和加速度数据的获取和显示，网页 demos-appliances.html 中包含如图 8.22 所示的 Java 脚本。

其中标记轮询函数 com.TI.tokenPoller()包含 3 个参数。

第 1 个参数 paramPage（参数页）的值为 param_demos.html，内容如图 8.23 所示。

其中定义了下列 3 个用户 GET 标记。

- __SL_G_UTP：设备温度（Device_Temperature）。

```
<script type="text/javascript">
$(function() {
    var Device_Temperature = $('#Device_Temperature_out'),
        Device_Accelerometer = $('#Device_Accelerometer_out'),
    demoPolling = new com.TI.tokenPoller({
        "paramPage": "param_demos.html",
        "refreshRate": 200,
        "valueMap": [{
            "paramId": "Device_Temperature",
            "outputSuccess": function(output) {
                Device_Temperature.html(output);
            },
            "outputDefault": function() {
                Device_Temperature.html("<i>reloading</i>");
            }
        },
        {
            "paramId": "Device_Accelerometer",
            "outputSuccess": function(output) {
                Device_Accelerometer.html(output);
            },
            "outputDefault": function() {
                Device_Accelerometer.html("<i>reloading</i>");
            }
        }]
    });
});
</script>
```

图 8.22　网页 demos-appliances.html 中温度和加速度获取和显示 Java 脚本

```
<html>
<body>
<p id="Device_Temperature">__SL_G_UTP</p>
<p id="Device_Accelerometer">__SL_G_UAC</p>
<p id="Device_LED_Status">__SL_G_ULD</p>
</body>
</html>
```

图 8.23　网页 param_demos.html 内容

- __SL_G_UAC：设备加速度（Device_Accelerometer）。
- __SL_G_ULD：设备 LED 状态（Device_LED_Status）（未使用）。

第 2 个参数 refreshRate（刷新率）的数值为 200，表示轮询间隔 200 ms。

第 3 个参数 valueMap（数值图）包含 Device_Temperature（设备温度）和 Device_Accelerometer（设备加速度）两部分，每部分都包含下列 3 个参数。

- paramId：参数标识，Device_Temperature 或 Device_Accelerometer。
- outputSuccess（成功输出）：在网页输出“水温”或“烘干机状态”。
- outputDefault（默认输出）：在网页输出“reloading”（重装）。

标记轮询函数 com.TI.tokenPoller()的代码在 js/com.TI.tokenPoller.js 文件中，其中的核心内容如图 8.24 所示。

其中$.ajax()方法通过 HTTP 请求加载远程数据，$.ajax()主要包含 4 个参数。

第 1 个参数 type 指定 HTTP 请求类型：GET 或 POST，默认为 GET。

```
function intervalFunc() {
    $.ajax({
        "type": "GET",
        "url": options.paramPage,
        "cache": false,
        "dataType": "html"
    }).done(function(data, status, xhr) {
        var i = 0,
        len = options.valueMap.length;
        if (status != 'success') {
            for (i=0;i<len;i++) {
                options.valueMap[i].outputDefault();
            }
        } else {
            var dataPage = $(data);
            for (i=0;i<len;i++) {
                options.valueMap[i].outputSuccess(dataPage.filter('#'+options.valueMap[i].paramId).text());
            }
        }
        invervalRunning = false;
    });
}
```

图 8.24　文件 com.TI.tokenPoller .js 中 com.TI.tokenPoller()函数核心内容

第 2 个参数 url 指定 HTTP 请求地址：options.paramPage 的值为 param_demos.html。

第 3 个参数 cache 指定是否从缓存中加载数据：true 或 false，默认为 true。

第 4 个参数 dataType 指定 HTTP 请求数据类型：html 为纯文本 HTML 信息。

$.ajax()方法的回调函数 done()包含 3 个参数。

第 1 个参数 data 指定 HTTP 请求返回的数据，内容如图 8.25 所示。

```
<html>
<body>
<p id="Device_Temperature">78 </p>
<p id="Device_Accelerometer">Stopped</p>
<p id="Device_LED_Status">Unknown Token</p>
</body>
</html>
```

图 8.25　$.ajax()方法回调函数 done()参数 data 内容

对比图 8.23 可以看出，HTTP 请求按用户 GET 标记返回对应的数据。

第 2 个参数 status 指定 HTTP 请求状态：success（成功）。

第 3 个参数 xhr 为 XMLHttpRequest 对象。

回调函数 done()对 HTTP 请求的返回数据做如下处理：如果 HTTP 请求不成功(status != 'success')，则循环 len(2)次 outputDefault（默认输出）“reloading”（重装）；如果 HTTP 请求成功(status = 'success')，则循环 len(2)次依次 outputSuccess（成功输出）从返回数据中过滤（filter）出的“Device_Temperature”和“Device_Accelerometer”的结果“78”(缺少单位“F”)和“Stopped”。

函数 intervalFunc()通过 setInterval()每 200 ms 调用一次，即网页 demos-appliances.html 中 Water Temp（水温）和 Dryer（烘干机状态）每 200 ms 更新一次。

HTTP 服务器接收 HTTP 请求后，对用户 GET 标记调用回调函数 GET 事件处理程序进行处理：对“__SL_G_UTP”标记，首先调用 TMP006DrvGetTemp()获得温度值（cTemp=78），然后调用 itoa()将温度值转换成字符串（“78”），并在温度字符串后添加空格（' '）和单位（'F'：华氏度），最后把字符串长度（sTempLen=3，应为 4）赋值给 ResponseData（响应数据）token_value（标记值）len（长度）；对“__SL_G_UAC”标记，首先调用 ReadAccSensor()判断加速度数据是否变化，然后根据加速度数据是否变化，把“Running”或“Stopped”赋值给 ResponseData（响应数据）token_value（标记值）data（数据），把对应的字符串长度赋值给 ResponseData（响应数据）token_value（标记值）len（长度）。

8.3.4 安全保障（Safety and Security）网页分析

安全保障（Safety and Security）演示功能可以看成家庭自动化（Home Automation）和家用电器（Home Appliances）的结合：用报警开关控制报警的设置，用 GET 方法获取开发板上加速度传感器的结果并控制报警显示，用 POST 方法控制开发板上 LED 的显示。

网页 demos_security.html 中报警开关和图像显示对应的 HTML 代码如图 8.26 所示。

```
<div class="switchControl" id="alarmSwitch">
    <div class="sprinklerOn">On</div>
    <div id="switchImage"></div>
    <div class="sprinklerOff">Off</div>
</div>
<img src="images/demo-security-alarmOff.jpg" name="alarmImage" id="alarmImage" />
```

图 8.26 网页 demos-security.html 中报警开关和图像显示对应 HTML 代码

当单击网页上的“alarmSwitch”（报警开关）时，调用 alarmSwitch.click()函数，alarm Switch.click()函数的内容如图 8.27 所示。

```
alarmSwitch.click(function(){
    if (switchBtn.hasClass('on')) {
        switchBtn.removeClass('on');
    } else {
        switchBtn.addClass('on');
    }
});
```

图 8.27 网页 demos-security.html 中 alarmSwitch.click()函数

函数 alarmSwitch.click()首先判断 switchBtn 是否有“on”类（报警开关状态），如果有说明报警开关原来是“ON”，则删除 switchBtn“on”类，否则添加 switchBtn“on”类。

为了实现加速度数据的获取和报警显示，网页 demos-security.html 中包含如图 8.28 所示的标记轮询函数 com.TI.tokenPoller()。

与家用电器（Home Appliances）不同，安全保障（Safety and Security）标记轮询函数 com.TI.tokenPoller()的第 3 个参数 valueMap（数值图）只包含 Device_Accelerometer（设备加速度），而且 outputSuccess（成功输出）和 outputDefault（默认输出）的内容不同。

成功输出（outputSuccess）包含下列 3 种情况（后两种情况可以合并）：如果报警开关“ON”并且加速度传感器的数据有变化，则网页报警显示为“images/ demo-security-alarmon.jpg”，

同时通过 com.TI.toggleLED()控制开发板上的 LED 点亮；如果报警开关“ON”但加速度传感器的数据没变化，则网页报警显示为“images/ demo-security-alarmoff.jpg”，同时通过 com.TI.toggleLED()控制开发板上的 LED 熄灭；如果报警开关“OFF”，则不管加速度传感器的数据有无变化，网页报警均显示为“images/demo-security-alarmoff.jpg”，同时通过 com.TI.toggleLED()控制 LED 熄灭。

默认输出（outputDefault）的结果和成功输出（outputSuccess）的后两种情况相同。

```
demoPolling = new com.TI.tokenPoller({
    "paramPage": "param_demos.html",
    "refreshRate": 200,
    "valueMap": [{
        "paramId": "Device_Accelerometer",
        "outputSuccess": function(output) {
            if (switchBtn.hasClass('on')) {
                if (output == "Running") {
                    alarmImage.attr('src',"images/demo-security-alarmOn.jpg");
                    com.TI.toggleLED('_ON');
                } else {
                    alarmImage.attr('src',"images/demo-security-alarmOff.jpg");
                    com.TI.toggleLED('_OFF');
                }
            } else {
                alarmImage.attr('src',"images/demo-security-alarmOff.jpg");
                com.TI.toggleLED('_OFF');
            }
        },
        "outputDefault": function() {
            alarmImage.attr('src',"images/demo-security-alarmOff.jpg");
            com.TI.toggleLED('_OFF');
        }
    }]
});
```

图 8.28　网页 demos-security.html 中 com.TI.tokenPoller()函数

8.3.5　智能能源网页分析

智能能源（Smart Energy）演示功能在家用电器（Home Appliances）温度显示的基础上，增加了数字步进器（Numeric Stepper）控制。

网页 demos_energy.html 中数字步进器和温度显示对应的 HTML 代码如图 8.29 所示。

```
<p for="slider-0"><strong>Desired Temperature (F):</strong></p>
<div id="tempInput"></div>
<p><strong>Current Temperature:</strong> <span id="Device_Temperature_out"><i>reloading</i></span></p>
```

图 8.29　网页 demos-energy.html 中数字步进器和温度显示对应 HTML 代码

为了实现温度数据的获取，网页 demos-energy.html 中包含如图 8.30 所示的标记轮询函数 com.TI.tokenPoller()。

智能能源（Smart Energy）标记轮询函数 com.TI.tokenPoller()的第 3 个参数 valueMap（数值图）只包含 Device_Temperature（设备温度），其中 outputSuccess（成功输出）的处理过程如下。

```
var demoPolling = new com.TI.tokenPoller({
    "paramPage": "param_demos.html",
    "refreshRate": 500,
    "valueMap": [{
        "paramId": "Device_Temperature",
        "outputSuccess": function(output) {
            if (!lastTemp) {
                lastTemp = parseInt(output);
            } else {
                lastTemp += dampenTempChange(lastTemp, output);
            }
            var temp = parseInt(lastTemp),
                newState = '_OFF';
            Device_Temperature.html(temp);
            if (lastInput < temp) {
                newState = '_Blink';
            } else if (lastInput > temp) {
                newState = '_ON';
            } else {
                newState = '_OFF';
            }
            if (newState != lastState) {
                lastState = newState;
                com.TI.toggleLED(newState);
            }
        },
        "outputDefault": function() {
            Device_Temperature.html("<i>reloading</i>");
        }
    }]
});
```

图 8.30　网页 demos-energy.html 中 com.TI.tokenPoller()函数

（1）如果最后温度（lastTemp）为空，则将 GET 的温度字符串通过调用 parseInt()转换为整数赋值给最后温度（lastTemp），否则通过调用 dampenTempChange()对温度值进行修正并在网页显示。

（2）如果所需温度（lastInput）小于当前温度（temp），则将新状态（newState）设为“_Blink”；如果大于则设为“_ON”，等于设为“_OFF”。

（3）如果新状态（newState）和最后状态（lastState）不同，则将新状态（newState）赋值给最后状态（lastState），并用新状态（newState）控制开发板上 LED 的显示。

8.4　HTTP 服务器小结

通过对 Provisioning_ap 示例程序和 Out_of_box 示例程序的分析可以看出：用户和主机之间的数据传输通过 GET 请求和 POST 请求实现，GET 请求通过网页和用户 GET 标记可以获取主机中的数据，POST 请求通过网页和用户 POST 标记可以将数据发送到主机。

CC3200 HTTP 服务器的工作原理如图 8.31 所示。

GET 请求有下列两种处理方法。

（1）在请求网页中直接包含用户 GET 标记。当用户通过浏览器浏览网页时，服务器生成 SL_NETAPP_HTTPGETTOKENVALUE_EVENT 事件，主机调用事件处理程序将 GET 标记值

返回给服务器，服务器将 GET 标记值放在网页中返回给用户，参见 Provisioning_ap 示例程序中“__SL_G_US1～5”的实现

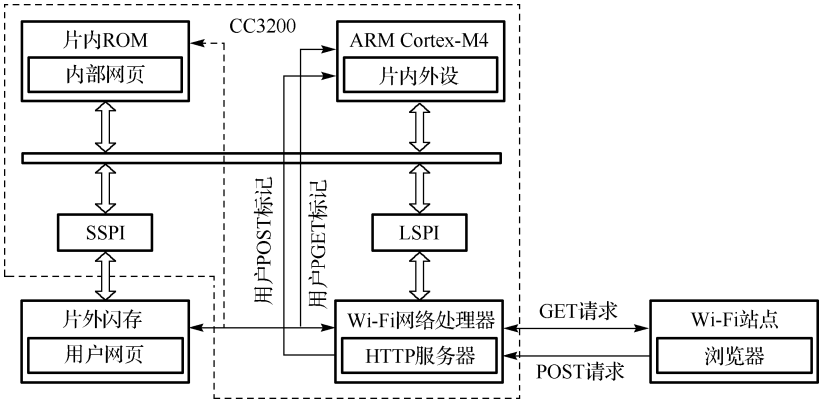


图 8.31　CC3200 HTTP 服务器工作原理

（2）在$.ajax()参数网页中间接包含用户 GET 标记。当用户通过浏览器浏览网页时，服务器执行 Java 脚本程序并调用$.ajax()，$.ajax()的 url 参数指定包含用户 GET 标记的参数网页（参见 Provisioning_ap 示例程序中的 param_status.html 和 Out_of_box 示例程序中的 param_demos.html），服务器生成 SL_NETAPP_HTTPGETTOKENVALUE_EVENT 事件，主机调用事件处理程序将 GET 标记值返回给服务器，服务器将 GET 标记值放在参数网页中返回给$.ajax()回调函数，回调函数对 GET 标记值处理后再将结果放在网页中返回给用户。参见 Provisioning_ap 示例程序中“__SL_G_US0”，以及 Out_of_box 示例程序中“__SL_G_UTP”和“__SL_G_UAC”的实现。

POST 请求有下列两种处理方法。

（1）在请求网页的表单中直接包含用户 POST 标记。表单提交后，服务器生成 SL_NETAPP_HTTPPOSTTOKENVALUE_EVENT 事件，主机调用事件处理程序对 POST 标记值进行处理，参见 Provisioning_ap 示例程序中“__SL_P_USC～G”的实现。

（2）在$.post()参数中间接包含用户 POST 标记。当服务器执行 Java 脚本程序并调用$.post()时，生成 SL_NETAPP_HTTPPOSTTOKENVALUE_EVENT 事件，主机调用事件处理程序对 POST 标记值进行处理，参见 Out_of_box 示例程序中“__SL_P_ULD”的实现。

GET 请求和 POST 请求的第一种实现方法只适合不对标记值进行处理的情况，相对比较简单。如果需要对标记值进行处理，只能使用第二种实现方法。

Java 脚本通过 setInterval()方法每隔指定时间调用$.ajax()或$.post()实现 GET 标记值的获取或 POST 标记值的传送。

第 9 章

电 源 管 理

CC3200 电源管理除了支持常规的睡眠（Sleep）和深睡眠（Deepsleep）低功耗模式外，还支持功耗更低的高级低功耗模式：低功耗深睡眠（LPDS）和休眠（Hibernate），两种模式下的工作电流分别是 700 μA（空闲连接）和 4 μA。

9.1 Sleep_deepsleep 示例程序分析

Sleep_deepsleep 示例程序在“example\sleep_deepsleep”目录中，Sleep_deepsleep 项目包含下列 6 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 UART 引脚配置程序文件。
- uart_if.c：UART 接口驱动程序文件。
- timer_if.c：GPT 驱动程序文件。
- wdt_if.c：WDT 驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Sleep_deepsleep 项目程序流程如图 9.1 所示。

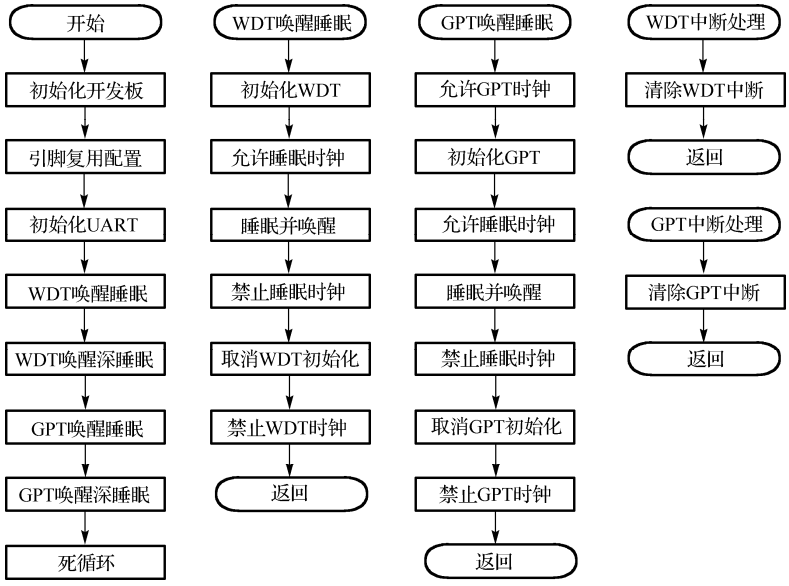


图 9.1　Sleep_deepsleep 项目程序流程

注意：建议用 UniFlash 将“example\sleep_deepsleep\ccs\Release”中的 sleep_deepsleep.bin 文件写入到开发板的串行闪存中（参考 2.2.2 节）对程序进行测试。CC3200 进入低功耗模式后将断开与调试器的连接，因此 CCS 不能对后续的操作进行调试。

Sleep_deepsleep 项目运行时在超级终端中显示下列信息。

```
SLEEP_DEEPSLEEP: Test Begin                睡眠深睡眠：测试开始
WDT_SLEEP: Entering Sleep                  WDT 睡眠：进入睡眠
WDT Interrupt occurred                     WDT 中断发生
WDT_SLEEP: Exiting Sleep                   WDT 睡眠：退出睡眠
WDT_DEEPSLEEP: Entering Deep Sleep         WDT 深睡眠：进入深睡眠
WDT Interrupt occured                      WDT 中断发生
WDT_DEEPSLEEP: Exiting Deep Sleep          WDT 深睡眠：退出深睡眠
GPT_SLEEP: Entering Sleep                  GPT 睡眠：进入睡眠
GPT TimerA0 Interrupt occurred             定时器 A0 中断发生
GPT_SLEEP: Exiting Sleep                   GPT 睡眠：退出睡眠
GPT_DEEPSLEEP: Entering Deep Sleep         GPT 深睡眠：进入深睡眠
GPT TimerA0 Interrupt occured              定时器 A0 中断发生
GPT_DEEPSLEEP: Exiting Deep Sleep          GPT 深睡眠：退出深睡眠
SLEEP_DEEPSLEEP: Test Complete             睡眠深睡眠：测试完成
```

拔掉 CC3200 LaunchPad 开发板 J12 上的短路块，串入电流表（万用表 200 mA 电流档），可测得 CC3200 的工作电流如表 9.1 所示。

表 9.1　CC3200 工作电流

电源模式	活动模式	睡眠模式	深睡眠模式
工作电流/mA	14	11	8

下面简要介绍 Sleep_deepsleep 项目的分析，包括 WDT 唤醒睡眠 PerformPRCMSleep WDTWakeup()分析和 GPT 唤醒睡眠 PerformPRCMSleepGPTWakeup()分析。

9.1.1　WDT 唤醒睡眠 PerformPRCMSleepWDTWakeup()分析

WDT 唤醒睡眠 PerformPRCMSleepWDTWakeup()包括初始化 WDT、允许 WDT 睡眠模式时钟、睡眠并唤醒、禁止 WDT 睡眠模式时钟、取消 WDT 初始化和禁止 WDT 时钟等。

1．初始化 WDT

初始化 WDT 的函数格式为

```
void WDT_IF_Init(fAPPWDTDevCallbk fpAppWDTCB, unsigned int uiReloadVal);
```

功能：初始化 WDT。

参数说明：

- fpAppWDTCB：WDT 中断处理程序，传递值是 AppWDTCallBackHandler。
- uiReloadVal：WDT 重装值，传递值是 4×SYS_CLK（80 MHz），定时 4 s。

WDT_IF_Init()的声明在 wdt_if.h 中，代码在 wdt_if.c 中。初始化 WDT 包括允许 WDT 运行模式时钟、解锁 WDT、注册 WDT 中断、设置 WDT 重装值和允许 WDT 等。

2．允许 WDT 睡眠模式时钟

允许 WDT 睡眠模式时钟的函数格式为

```
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
```

功能：允许外设时钟。

参数说明：

- ulPeripheral：设备号，在 prcm.h 中定义。
- ulClkFlags：时钟标志，设置值是 PRCM_SLP_MODE_CLK（睡眠模式时钟），包括下列值。
 - ✧ PRCM_RUN_MODE_CLK：运行模式时钟，0x0000 0001。
 - ✧ PRCM_SLP_MODE_CLK：睡眠模式时钟，0x0000 0100。
 - ✧ PRCM_DSLP_MODE_CLK：深睡眠模式时钟，0x0001 0000。

PRCMPeripheralClkEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

3．睡眠并唤醒

进入睡眠模式的函数格式为

```
void PRCMSleepEnter(void);
```

功能：进入睡眠模式。

任何可用中断都可以唤醒睡眠，唤醒睡眠后返回调用函数。

PRCMSleepEnter()的声明在 prcm.h 中，代码在 prcm.c 中。唤醒睡眠通过 WDT 中断实现，在 WDT 中断处理程序中通过调用 WatchdogIntClear()清除 WDT 中断。

4．禁止 WDT 睡眠模式时钟

禁止 WDT 睡眠模式时钟的函数格式为

```
void PRCMPeripheralClkDisable(unsigned long ulPeripheral,
                              unsigned long ulClkFlags);
```

功能：禁止外设时钟。

参数说明：

- ulPeripheral：设备号，在 prcm.h 中定义。
- ulClkFlags：时钟标志，设置值是 PRCM_SLP_MODE_CLK（睡眠模式时钟），包括下列值。
 - ✧ PRCM_RUN_MODE_CLK：运行模式时钟，0x0000 0001。
 - ✧ PRCM_SLP_MODE_CLK：睡眠模式时钟，0x0000 0100。
 - ✧ PRCM_DSLP_MODE_CLK：深睡眠模式时钟，0x0001 0000。

PRCMPeripheralClkDisable()的声明在 prcm.h 中，代码在 prcm.c 中。

5．取消 WDT 初始化

取消 WDT 初始化的函数格式为

```
void WDT_IF_DeInit(void);
```

功能：取消 WDT 初始化。

WDT_IF_DeInit()的声明在 wdt_if.h 中，代码在 wdt_if.c 中。取消 WDT 初始化包括解锁 WDT、清除 WDT 中断和注销 WDT 中断等。

6．禁止 WDT 时钟

禁止 WDT 时钟通过调用 PRCMPeripheralClkDisable()禁止 WDT 运行模式时钟 PRCM_RUN_MODE_CLK。

WDT 唤醒深睡眠 PerformPRCMDeepSleepWDTWakeup()和 WDT 唤醒睡眠 Perform PRCMSleepWDTWakeup()的过程类似，不同之处如下。

（1）允许 WDT 深睡眠模式时钟 PRCM_DSLP_MODE_CLK。

（2）调用 PRCMDeepSleepEnter()进入深睡眠。进入深睡眠模式的函数格式为

```
void PRCMDeepSleepEnter(void);
```

功能：进入深睡眠模式。

PRCMDeepSleepEnter()的声明在 prcm.h 中，代码在 prcm.c 中。

（3）禁止 WDT 深睡眠模式时钟 PRCM_DSLP_MODE_CLK。

9.1.2 GPT 唤醒睡眠 PerformPRCMSleepGPTWakeup()分析

GPT 唤醒睡眠 PerformPRCMSleepGPTWakeup()包括允许 GPT 时钟、初始化 GPT、允许 GPT 睡眠模式时钟、睡眠并唤醒、禁止 GPT 睡眠模式时钟、取消 GPT 初始化和禁止 GPT 时钟等。

1．允许 GPT 时钟

允许 GPT 时钟通过调用 PRCMPeripheralClkEnable()允许 TIMERA0 的运行模式时钟 PRCM_RUN_MODE_CLK。

2．初始化 GPT

初始化 GPT 包括初始化定时器 Timer_IF_Init()、设置定时器中断 Timer_IF_IntSetup()和启动定时器 Timer_IF_Start()等。

（1）初始化定时器 Timer_IF_Init()。初始化定时器 Timer_IF_Init()的函数格式为

```
void Timer_IF_Init(unsigned long ePeripheral, unsigned long ulBase,
                   unsigned long ulConfig, unsigned long ulTimer,
                   unsigned long ulValue);
```

功能：初始化定时器。

参数说明：

- ePeripheral：设备号，在 prcm.h 中定义。
- ulBase：定时器基地址，在 hw_memmap.h 中定义。
- ulConfig：定时器配置，设置值是 TIMER_CFG_ONE_SHOT（32 位单次定时器）。
- ulTimer：定时器选择，设置值是 TIMER_BOTH（定时器 A 和定时器 B）。
- ulValue：预分频值，0～255（1～256 分频），设置值是 0（1 分频即不分频）。

Timer_IF_Init()的声明在 timer_if.h 中，代码在 timer_if.c 中。Timer_IF_Init()依次调用 PRCMPeripheralClkEnable()、PRCMPeripheralReset()、Timer Configure()和 TimerPrescaleSet()等实现定时器初始化。

（2）设置定时器中断 Timer_IF_IntSetup()。设置定时器中断 Timer_IF_IntSetup()的函数格式为

```
void Timer_IF_IntSetup(unsigned long ulBase, unsigned long ulTimer,
                       void (*TimerBaseIntHandler)(void));
```

功能：设置定时器中断。

参数说明：

- ulBase：定时器基地址，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，设置值是 TIMER_BOTH（定时器 A 和定时器 B）。
- *TimerBaseIntHandler：定时器中断处理程序指针，中断处理程序是 AppGPTCallBack Handler()。

Timer_IF_IntSetup()的声明在 timer_if.h 中，代码在 timer_if.c 中。Timer_IF_IntSetup()通过调用 TimerIntRegister()和 TimerIntEnable()实现中断处理程序的注册和允许。

（3）启动定时器 Timer_IF_Start()。启动定时器 Timer_IF_Start()的函数格式为

```
void Timer_IF_Start(unsigned long ulBase, unsigned long ulTimer,
                    unsigned long ulValue);
```

功能：启动定时器。

参数说明：

- ulBase：定时器基地址，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，设置值是 TIMER_BOTH（定时器 A 和定时器 B）。
- ulValue：定时器初值，设置值是 4×SYS_CLK（80 MHz），定时 4 s。

Timer_IF_Start()的声明在 timer_if.h 中，代码在 timer_if.c 中。Timer_IF_Start()通过调用 TimerLoadSet()和 TimerEnable()实现定时器的启动。

3．允许 GPT 睡眠模式时钟

允许 GPT 睡眠模式时钟通过调用 PRCMPeripheralClkEnable()允许 TIMERA0 的睡眠模式时钟 PRCM_SLP_MODE_CLK。

4．睡眠并唤醒

进入睡眠通过调用 PRCMSleepEnter()实现，睡眠唤醒通过 GPT 中断实现，在 GPT 中断处理程序中通过调用 TimerIntClear()清除 GPT 中断。

5．禁止 GPT 睡眠模式时钟

禁止 GPT 睡眠模式时钟通过调用 PRCMPeripheralClkDisable()禁止 TIMERA0 睡眠模式时钟 PRCM_SLP_MODE_CLK。

6．取消 GPT 初始化

取消 GPT 初始化包括停止定时器 Timer_IF_Stop()和取消定时器初始化 Timer_IF_DeInit()。

（1）停止定时器 Timer_IF_Stop()。停止定时器 Timer_IF_Stop()的函数格式为

```
void Timer_IF_Stop(unsigned long ulBase, unsigned long ulTimer);
```

功能：停止定时器。

参数说明：

- ulBase：定时器基地址，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，设置值是 TIMER_BOTH（定时器 A 和定时器 B）。

Timer_IF_Stop()的声明在 timer_if.h 中，代码在 timer_if.c 中，Timer_IF_Stop()通过调用 TimerDisable()实现定时器的停止。

（2）取消定时器初始化 Timer_IF_DeInit()。取消定时器初始化 Timer_IF_DeInit()的函数格式为

```
void Timer_IF_DeInit(unsigned long ulBase, unsigned long ulTimer);
```

功能：取消定时器初始化。

参数说明：

- ulBase：定时器基地址，在 hw_memmap.h 中定义。
- ulTimer：定时器选择，设置值是 TIMER_BOTH（定时器 A 和定时器 B）。

Timer_IF_DeInit()的声明在 timer_if.h 中，代码在 timer_if.c 中，Timer_IF_DeInit()通过调用 TimerIntDisable()和 TimerIntUnregister()实现中断处理程序禁止和注销。

7．禁止 GPT 时钟

禁止 GPT 时钟通过调用函数 PRCMPeripheralClkDisable()禁止 TIMERA0 的运行模式时钟 PRCM_RUN_MODE_CLK。

GPT 唤醒深睡眠和 GPT 唤醒睡眠的过程也类似，不同之处如下。

（1）允许 GPT 深睡眠模式时钟 PRCM_DSLP_MODE_CLK。

（2）调用 PRCMDeepSleepEnter()进入深睡眠。

（3）禁止 GPT 深睡眠模式时钟 PRCM_DSLP_MODE_CLK。

综上所述，Sleep_deepsleep 项目使用的与电源管理有关的 API 函数如表 9.2 所示。

表 9.2　Sleep_deepsleep 项目使用的与电源管理有关的 API 函数

功　　能	函数名	调用函数	代码文件
允许外设时钟	PRCMPeripheralClkEnable()	PerformPRCMSleepWDTWakeup()、PerformPRCMDeepSleepWDTWakeup()、PerformPRCMSleepGPTWakeup()、PerformPRCMDeepSleepGPTWakeup()	prcm.c
进入睡眠模式	PRCMSleepEnter()	PerformPRCMSleepWDTWakeup()、PerformPRCMSleepGPTWakeup()	prcm.c
进入深睡眠模式	PRCMDeepSleepEnter()	PerformPRCMDeepSleepWDTWakeup()、PerformPRCMDeepSleepGPTWakeup()	prcm.c
禁止外设时钟	PRCMPeripheralClkDisable()	PerformPRCMSleepWDTWakeup()、PerformPRCMDeepSleepWDTWakeup()、PerformPRCMSleepGPTWakeup()、PerformPRCMDeepSleepGPTWakeup()	prcm.c

9.2　Deepsleep_nw 示例程序分析

Deepsleep_nw 示例程序在“example\deepsleep_nw”目录中，Deepsleep_nw 项目包含下列 7 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO 和 UART 引脚配置程序文件。
- gpio_if.c：GPIO 接口驱动程序文件。
- uart_if.c：UART 接口驱动程序文件。
- timer_if.c：GPT 驱动程序文件。
- nw_if.c：网络驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Deepsleep_nw 项目程序流程如图 9.2 所示。

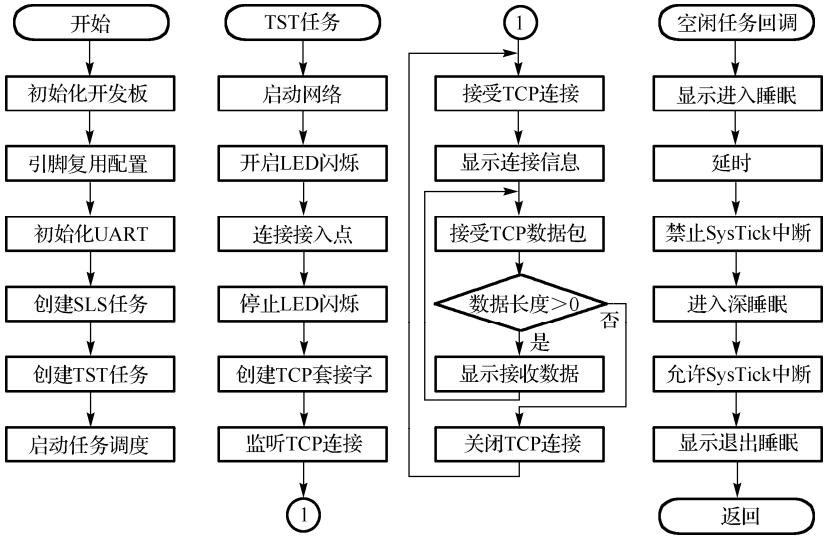


图 9.2　Deepsleep_nw 项目程序流程

注意：建议用 UniFlash 将“D:\TI\workspace_v6_1\deepsleep_nw\Release”中的 deepsleep_

nw.bin 文件（修改接入点信息后重新生成的文件）写入到开发板的串行闪存中（参考 2.2.2 节）对程序进行测试。CC3200 进入低功耗模式后将断开与调试器的连接，因此 CCS 不能对后续的操作进行调试。

Deepsleep_nw 项目运行时在超级终端中显示下列信息。

```
DEEPSLEEP: Test Begin                          深睡眠：测试开始
DEEPSLEEP: Entering DeepSleep                  深睡眠：进入深睡眠
Host Driver Version: 1.0.0.10
Build Version 2.0.7.0.31.0.0.4.1.1.5.3.3
DEEPSLEEP: Exiting DeepSleep                   深睡眠：退出深睡眠
DEEPSLEEP: Entering DeepSleep                  深睡眠：进入深睡眠
Device is configured in default state
Started SimpleLink Device: STA Mode
[WLAN EVENT] STA Connected to the AP: 1234 , BSSID: 3c:e5:a6:1:7:bc
Device has connected to 1234
Device IP Address is 192.168.1.6

DEEPSLEEP: Exiting DeepSleep                   深睡眠：退出深睡眠
DEEPSLEEP: Socket created                      深睡眠：创建套接字
DEEPSLEEP: Listening on Socket...              深睡眠：监听套接字
DEEPSLEEP: Waiting for client.......           深睡眠：等待客户端连接
```

在安卓手机上打开 Wi-Fi，连接用户接入点“1234”，运行 iperf，在 iPerf 命令行参数（iPerf command line arguments）框中输入下列命令行参数。

```
-c 192.168.1.6
```

单击“关”按钮，按钮变为“开”，表示打开客户端模式，通过 IP 地址建立与设备的无线连接并发送数据，设备在超级终端中显示下列信息。

```
DEEPSLEEP: Connected to client [0xc0a80109]深睡眠：连接到客户端 192.168.1.9
DEEPSLEEP: received 512 bytes                  深睡眠：接收 512B
DEEPSLEEP: received message :                  深睡眠：接收信息：
```

拔掉 CC3200 LaunchPad 开发板 J12 上的短路块，串入电流表（万用表 200 mA 电流挡），可测得 CC3200 的工作电流如表 9.3 所示。

表 9.3 CC3200 工作电流

工作模式	活动模式	深睡眠模式	接收模式
工作电流/mA	16	8	20～60

Deepsleep_nw 项目的主要内容包括网络操作和电源管理两部分。网络操作部分的主要内容是启动网络、连接接入点、创建 TCP 套接字、监听 TCP 连接、接受 TCP 连接和接收 TCP 数据包等，具体内容和前面的介绍类似，这里不再赘述。电源管理部分比较简单，主要是利用 nw_if.c 中的空闲任务回调函数 vApplicationIdle Hook()实现深睡眠操作，程序代码如图 9.3 所示。

其中进入深睡眠函数 PRCMDeepSleepEnter()实现深睡眠操作，将 nw_if.c 中空闲任务回调函数 vApplicationIdleook()的内容注释掉，Deepsleep_nw 项目就变成一般的网络操作项目。

```
128 void
129 vApplicationIdleHook( void)
130 {
131     int iRetVal;
132
133     //
134     // Enter SLEEP...WaitForInterrupt ARM intrinsic
135     //
136     DBG_PRINT("DEEPSLEEP: Entering DeepSleep\n\r");
137
138     //MAP_UtilsDelay(80000);
139     for(iRetVal = 0; iRetVal < 80000; iRetVal++);
140
141     //
142     // Disable the SYSTICK interrupt
143     //
144     MAP_IntDisable(FAULT_SYSTICK);
145     MAP_PRCMDeepSleepEnter();
146
147     //
148     // Enable the SYSTICK interrupt
149     //
150     MAP_IntEnable(FAULT_SYSTICK);
151     DBG_PRINT("DEEPSLEEP: Exiting DeepSleep\n\r");
152 }
```

图 9.3　Deepsleep_nw 项目空闲任务回调函数

9.3　Hib 示例程序分析

Hib 示例程序在“example\hib”目录中，Hib 项目包含下列 8 个程序文件。

- main.c：主程序文件。
- pinmux.c：PinMux 应用程序生成的 GPIO 和 UART 引脚配置程序文件。
- gpio_if.c：GPIO 接口驱动程序文件。
- uart_if.c：UART 接口驱动程序文件。
- timer_if.c：GPT 驱动程序文件。
- utils_if.c：串行闪存掉电程序文件。
- network_if.c：网络驱动程序文件。
- startup_ccs.c：CCS 启动程序文件。

Hib 项目程序流程如图 9.4 所示。

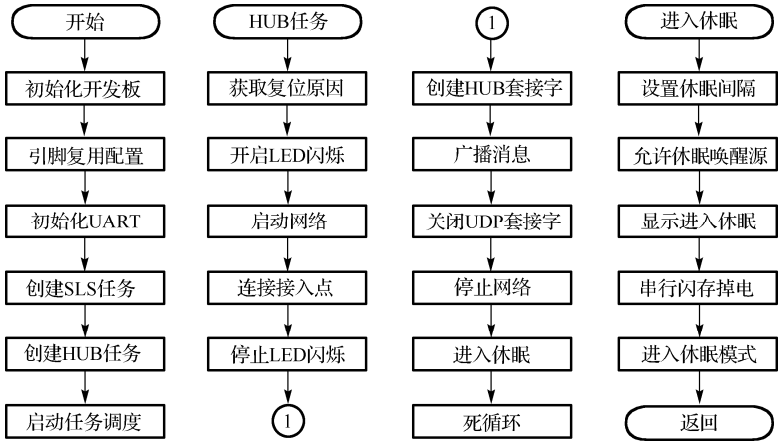


图 9.4　Hib 项目程序流程

注意：建议用 UniFlash 将“D:\TI\workspace_v6_1\hib\Release”中的 hib.bin 文件（修改接入点信息后重新生成的文件）写入到开发板的串行闪存中（参考 2.2.2 节）对程序进行测试。CC3200 进入休眠模式后将断开与调试器的连接，因此 CCS 不能对后续的操作进行调试。

Hib 项目运行时在超级终端中显示下列信息。

```
    HIB: Wake up on Power ON                    休眠：上电唤醒
    Host Driver Version: 1.0.0.10
    Build Version 2.0.7.0.31.0.0.4.1.1.5.3.3
    Device is configured in default state
    Started SimpleLink Device: STA Mode
    [WLAN EVENT] STA Connected to the AP: 1234 , BSSID: 3c:e5:a6:1:7:bc
    [NETAPP EVENT] IP acquired by the device

    Device has connected to 1234
    Device IP Address is 192.168.1.6

    HIB: Socket created                         休眠：套接字已创建
    HIB: sent message                           休眠：发送消息
    HIB: Socket closed                          休眠：套接字已关闭
    SL Disconnect...
    [WLAN EVENT]Device disconnected from the AP:1234,BSSID:3c:e5:a6:1:7:bc on
application's request
    HIB: Entering HIBernate...                  休眠：进入休眠
    HIB: Woken up from Hibernate                休眠：休眠唤醒
```

拔掉 CC3200 LaunchPad 开发板 J12 上的短路块，串入电流表（万用表 200 mA 电流挡），可测得 CC3200 的工作电流如表 9.4 所示。

表 9.4　CC3200 工作电流

工作模式	活动模式	发送模式	休眠模式
工作电流/mA	16	20～106	0

Hib 项目的主要内容包括网络操作和电源管理两部分。

1．网络操作

网络操作部分的主要内容是获取复位原因、启动网络、连接接入点、创建 UDP 套接字、广播消息、关闭 UDP 套接字和停止网络等，除了获取复位原因，其他内容和前面的介绍类似，这里不再赘述。

获取复位原因 PRCMSysResetCauseGet()的函数格式为

```
    unsigned long PRCMSysResetCauseGet();
```

功能：获取系统复位原因。

返回值：系统复位原因，包括下列值。

- PRCM_POWER_ON：上电复位，0。
- PRCM_LPDS_EXIT：退出 LPDS，1。

- PRCM_CORE_RESET：内核复位，3。
- PRCM_MCU_RESET：MCU 复位，4。
- PRCM_WDT_RESET：WDT 复位，5。
- PRCM_SOC_RESET：SOC 复位，6。
- PRCM_HIB_EXIT：退出休眠，7。

PRCMSysResetCauseGet()的声明在 prcm.h 中，代码在 prcm.c 中。系统第一次是 PRCM_POWER_ON（上电复位），程序在超级终端中显示标题和下列信息。

```
HIB: Wake up on Power ON                    休眠：上电唤醒
```

系统休眠唤醒后是 PRCM_HIB_EXIT（退出休眠），程序在超级终端中显示下列信息。

```
HIB: Woken up from Hibernate                休眠：休眠唤醒
```

2．电源管理

电源管理部分的主要内容是设置休眠间隔、允许休眠唤醒源和进入休眠模式等，程序代码如图 9.5 所示。

```
394 void EnterHIBernate()
395 {
396 #define SLOW_CLK_FREQ           (32*1024)
397     //
398     // Configure the HIB module RTC wake time
399     //
400     MAP_PRCMHibernateIntervalSet(5 * SLOW_CLK_FREQ);
401
402     //
403     // Enable the HIB RTC
404     //
405     MAP_PRCMHibernateWakeupSourceEnable(PRCM_HIB_SLOW_CLK_CTR);
406
407     DBG_PRINT("HIB: Entering HIBernate...\n\r");
408     MAP_UtilsDelay(80000);
409
410     //
411     // powering down SPI Flash to save power
412     //
413     Utils_SpiFlashDeepPowerDown();
414     //
415     // Enter HIBernate mode
416     //
417     MAP_PRCMHibernateEnter();
418 }
```

图 9.5　Hib 项目进入休眠函数

（1）设置休眠间隔。设置休眠间隔 PRCMHibernateIntervalSet()的函数格式为

```
void PRCMHibernateIntervalSet(unsigned long long ullTicks);
```

功能：设置休眠间隔。

参数说明：ullTicks，32.768 kHz 时钟数，传递值是 5×32×1024（5 s）。

PRCM HibernateIntervalSet()的声明在 prcm.h 中，代码在 prcm.c 中。

（2）允许休眠唤醒源。允许休眠唤醒源 PRCMHibernateWakeupSourceEnable()的函数格式为

```
void PRCMHibernateWakeupSourceEnable(unsigned long ulHIBWakupSrc);
```

功能：允许休眠唤醒源。

参数说明：ulHIBWakupSrc，休眠唤醒源，设置值是 PRCM_HIB_SLOW_CLK_CTR（慢时钟定时器），包含下列值。

- PRCM_HIB_SLOW_CLK_CTR：慢时钟定时器，0x0000 0001。
- PRCM_HIB_GPIO2：GPIO2，0x0001 0000。
- PRCM_HIB_GPIO4：GPIO4，0x0002 0000。
- PRCM_HIB_GPIO13：GPIO13，0x0004 0000。
- PRCM_HIB_GPIO17：GPIO17，0x0008 0000。
- PRCM_HIB_GPIO11：GPIO11，0x0010 0000。
- PRCM_HIB_GPIO24：GPIO24，0x0020 0000。
- PRCM_HIB_GPIO26：GPIO26，0x0040 0000。

PRCMHibernateWakeupSourceEnable()的声明在 prcm.h 中，代码在 prcm.c 中。

（3）进入休眠模式。进入休眠模式 PRCMHibernateEnter()的函数格式为

```
void PRCMHibernateEnter();
```

功能：进入休眠模式。

允许的休眠唤醒源可以唤醒休眠，唤醒休眠后复位设备，从头执行程序，因此进入休眠模式函数不会返回。

PRCMHibernateEnter()的声明在 prcm.h 中，代码在 prcm.c 中。

综上所述，Hib 项目使用的与电源管理有关的 API 函数如表 9.5 所示。

表 9.5　Deepsleep_nw 项目使用的与电源管理有关的 API 函数

功　能	函数名	调用函数	代码文件
设置休眠间隔	PRCMHibernateIntervalSet()	EnterHibernate()	prcm.c
允许休眠唤醒源	PRCMHibernateWakeupSourceEnable()	EnterHibernate()	prcm.c
进入休眠模式	PRCMHibernateEnter()	EnterHibernate()	prcm.c

低功耗深睡眠（LPDS）的操作方法和休眠（Hibernate）类似，读者可以参考 Hib 示例程序自行修改实现。

附录 A

CC3200 外设驱动 API

1．GPIO：14 个

```
void GPIODirModeSet(unsigned long ulPort, unsigned char ucPins,
                    unsigned long ulPinIO);
unsigned long GPIODirModeGet   (unsigned long ulPort, unsigned char ucPin);
void GPIOPinWrite(unsigned long ulPort, unsigned char ucPins,
                  unsigned char ucVal);
long GPIOPinRead(unsigned long ulPort, unsigned char ucPins);
void GPIOIntTypeSet(unsigned long ulPort, unsigned char ucPins,
                    unsigned long ulIntType);
unsigned long GPIOIntTypeGet(unsigned long ulPort, unsigned char ucPin);
void GPIOIntRegister(unsigned long ulPort, void (*pfnHandler)(void));
void GPIOIntUnregister(unsigned long ulPort);
void GPIOIntEnable(unsigned long ulPort, unsigned long ulIntFlags);
void GPIOIntDisable(unsigned long ulPort, unsigned long ulIntFlags);
long GPIOIntStatus(unsigned long ulPort, tBoolean bMasked);
void GPIOIntClear(unsigned long ulPort, unsigned long ulIntFlags);
void GPIODMATriggerEnable(unsigned long ulPort);
void GPIODMATriggerDisable(unsigned long ulPort);
```

2．UART：36 个

```
void UARTConfigSetExpClk(unsigned long ulBase, unsigned long ulUARTClk,
                         unsigned long ulBaud, unsigned long ulConfig);
void UARTConfigGetExpClk(unsigned long ulBase, unsigned long ulUARTClk,
                         unsigned long *pulBaud, unsigned long *pulConfig);
void UARTEnable(unsigned long ulBase);
void UARTDisable(unsigned long ulBase);
void UARTCharPut(unsigned long ulBase, unsigned char ucData);
tBoolean UARTCharPutNonBlocking(unsigned long ulBase, unsigned char ucData);
long UARTCharGet(unsigned long ulBase);
long UARTCharGetNonBlocking(unsigned long ulBase);
void UARTIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void UARTIntUnregister(unsigned long ulBase);
void UARTIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
void UARTIntDisable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long UARTIntStatus(unsigned long ulBase, tBoolean bMasked);
void UARTIntClear(unsigned long ulBase, unsigned long ulIntFlags);
```

```
void UARTDMAEnable(unsigned long ulBase, unsigned long ulDMAFlags);
void UARTDMADisable(unsigned long ulBase, unsigned long ulDMAFlags);
void UARTParityModeSet(unsigned long ulBase, unsigned long ulParity);
unsigned long UARTParityModeGet(unsigned long ulBase);
void UARTFIFOEnable(unsigned long ulBase);
void UARTFIFODisable(unsigned long ulBase);
void UARTFIFOLevelSet(unsigned long ulBase, unsigned long ulTxLevel,
                      unsigned long ulRxLevel);
void UARTFIFOLevelGet(unsigned long ulBase, unsigned long *pulTxLevel,
                      unsigned long *pulRxLevel);
tBoolean UARTCharsAvail(unsigned long ulBase);
tBoolean UARTSpaceAvail(unsigned long ulBase);
void UARTBreakCtl(unsigned long ulBase, tBoolean bBreakState);
tBoolean UARTBusy(unsigned long ulBase);
unsigned long UARTRxErrorGet(unsigned long ulBase);
void UARTRxErrorClear(unsigned long ulBase);
void UARTTxIntModeSet(unsigned long ulBase, unsigned long ulMode);
unsigned long UARTTxIntModeGet(unsigned long ulBase);
void UARTModemControlSet(unsigned long ulBase, unsigned long ulControl);
unsigned long UARTModemControlGet(unsigned long ulBase);
void UARTModemControlClear(unsigned long ulBase, unsigned long ulControl);
unsigned longUARTModemStatusGet(unsigned long ulBase);
void UARTFlowControlSet(unsigned long ulBase, unsigned long ulMode);
unsigned long UARTFlowControlGet(unsigned long ulBase);
```

3. SPI：24 个

```
void SPIConfigSetExpClk(unsigned long ulBase, unsigned long ulSPIClk,
                        unsigned long ulBitRate, unsigned long ulMode,
                        unsigned long ulSubMode, unsigned long ulConfig);
void SPIReset(unsigned long ulBase);
void SPIEnable(unsigned long ulBase);
void SPIDisable(unsigned long ulBase);
void SPIWordCountSet(unsigned long ulBase, unsigned long ulWordCount);
long SPITransfer(unsigned long ulBase, unsigned char *ucDout,
                 unsigned char *ucDin, unsigned long ulSize,
                 unsigned long ulFlags);  3.3(80)
void SPICSEnable(unsigned long ulBase);3.3(81)
void SPICSDisable(unsigned long ulBase);3.3(82)
void SPIDataPut(unsigned long ulBase, unsigned long ulData);3.3(81)
long SPIDataPutNonBlocking(unsigned long ulBase, unsigned long ulData);
void SPIDataGet(unsigned long ulBase, unsigned long *pulData);3.3(81)
long SPIDataGetNonBlocking(unsigned long ulBase, unsigned long * pulData);
void SPIIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void SPIIntUnregister(unsigned long ulBase);
void SPIIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
void SPIIntDisable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long SPIIntStatus(unsigned long ulBase, tBoolean bMasked);
```

```
void SPIIntClear(unsigned long ulBase, unsigned long ulIntFlags);
void SPIDmaEnable(unsigned long ulBase, unsigned long ulFlags);
void SPIDmaDisable(unsigned long ulBase, unsigned long ulFlags);
void SPIFIFOEnable(unsigned long ulBase, unsigned long ulFlags);
void SPIFIFODisable(unsigned long ulBase, unsigned long ulFlags);
void SPIFIFOLevelSet(unsigned long ulBase, unsigned long ulTxLevel,
                     unsigned long ulRxLevel);
void SPIFIFOLevelGet(unsigned long ulBase, unsigned long *pulTxLevel,
                     unsigned long *pulRxLevel);
```

4．I2C：53 个

```
void I2CMasterInitExpClk(uint32_t ui32Base, uint32_t ui32I2CClk,
                         bool bFast);
void I2CMasterEnable(uint32_t ui32Base);
void I2CMasterDisable(uint32_t ui32Base);
void I2CMasterSlaveAddrSet(uint32_t ui32Base, uint8_t ui8SlaveAddr,
                           bool bReceive);
void I2CMasterDataPut(uint32_t ui32Base, uint8_t ui8Data);
uint32_t I2CMasterDataGet(uint32_t ui32Base);
void I2CMasterTimeoutSet(uint32_t ui32Base, uint32_t ui32Value);
void I2CMasterControl(uint32_t ui32Base, uint32_t ui32Cmd);
uint32_t I2CMasterErr(uint32_t ui32Base);
void I2CIntRegister(uint32_t ui32Base, void(*pfnHandler)(void));
void I2CIntUnregister(uint32_t ui32Base);
void I2CMasterIntEnableEx(uint32_t ui32Base, uint32_t ui32IntFlags);
void I2CMasterIntDisableEx(uint32_t ui32Base, uint32_t ui32IntFlags);
uint32_t I2CMasterIntStatusEx(uint32_t ui32Base, bool bMasked);
void I2CMasterIntClearEx(uint32_t ui32Base, uint32_t ui32IntFlags);
void I2CMasterIntEnable(uint32_t ui32Base);
void I2CMasterIntDisable(uint32_t ui32Base);
bool I2CMasterIntStatus(uint32_t ui32Base, bool bMasked);
void I2CMasterIntClear(uint32_t ui32Base);
void I2CMasterBurstLengthSet(uint32_t ui32Base, uint8_t ui8Length);
uint32_t I2CMasterBurstCountGet(uint32_t ui32Base);
void I2CMasterGlitchFilterConfigSet(uint32_t ui32Base,
                                    uint32_t ui32Config);
uint32_t I2CMasterLineStateGet(uint32_t ui32Base);
bool I2CMasterBusBusy(uint32_t ui32Base);
bool I2CMasterBusy(uint32_t ui32Base);
void I2CSlaveEnable(uint32_t ui32Base);
void I2CSlaveDisable(uint32_t ui32Base);
void I2CSlaveInit(uint32_t ui32Base, uint8_t ui8SlaveAddr);
void I2CSlaveAddressSet(uint32_t ui32Base, uint8_t ui8AddrNum,
                        uint8_t ui8SlaveAddr);
uint32_t I2CSlaveStatus(uint32_t ui32Base);
void I2CSlaveDataPut(uint32_t ui32Base, uint8_t ui8Data);
uint32_t I2CSlaveDataGet(uint32_t ui32Base);
```

```
void I2CSlaveACKOverride(uint32_t ui32Base, bool bEnable);
void I2CSlaveACKValueSet(uint32_t ui32Base, bool bACK);
void I2CSlaveIntEnableEx(uint32_t ui32Base, uint32_t ui32IntFlags);
void I2CSlaveIntDisableEx(uint32_t ui32Base, uint32_t ui32IntFlags);
uint32_t I2CSlaveIntStatusEx(uint32_t ui32Base, bool bMasked);
void I2CSlaveIntClearEx(uint32_t ui32Base, uint32_t ui32IntFlags);
void I2CSlaveIntEnable(uint32_t ui32Base);
void I2CSlaveIntDisable(uint32_t ui32Base);
bool I2CSlaveIntStatus(uint32_t ui32Base, bool bMasked);
void I2CSlaveIntClear(uint32_t ui32Base);
void I2CSlaveFIFOEnable(uint32_t ui32Base, uint32_t ui32Config);
void I2CSlaveFIFODisable(uint32_t ui32Base);
void I2CFIFODataPut(uint32_t ui32Base, uint8_t ui8Data);
uint32_t I2CFIFODataPutNonBlocking(uint32_t ui32Base, uint8_t ui8Data);
uint32_t I2CFIFODataGet(uint32_t ui32Base);
uint32_t I2CFIFODataGetNonBlocking(uint32_t ui32Base, uint8_t *pui8Data);
uint32_t I2CFIFOStatus(uint32_t ui32Base);
void I2CTxFIFOConfigSet(uint32_t ui32Base, uint32_t ui32Config);
void I2CTxFIFOFlush(uint32_t ui32Base);
void I2CRxFIFOConfigSet(uint32_t ui32Base, uint32_t ui32Config);
void I2CRxFIFOFlush(uint32_t ui32Base);
```

5. GPT：23 个

```
void TimerConfigure(unsigned long ulBase, unsigned long ulConfig);
void TimerPrescaleSet(unsigned long ulBase, unsigned long ulTimer,
                      unsigned long ulValue);
unsigned long TimerPrescaleGet(unsigned long ulBase, unsigned long ulTimer);
void TimerPrescaleMatchSet(unsigned long ulBase, unsigned long ulTimer,
                           unsigned long ulValue);
unsigned long TimerPrescaleMatchGet(unsigned long ulBase,
                                    unsigned long ulTimer);
void TimerIntRegister(unsigned long ulBase, unsigned long ulTimer,
                      void (*pfnHandler)(void));
void TimerIntUnregister(unsigned long ulBase, unsigned long ulTimer);
void TimerIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
void TimerIntDisable(unsigned long ulBase, unsigned long ulIntFlags);
void TimerLoadSet(unsigned long ulBase, unsigned long ulTimer,
                  unsigned long ulValue);
unsigned long TimerLoadGet(unsigned long ulBase, unsigned long ulTimer);
void TimerEnable(unsigned long ulBase, unsigned long ulTimer);
void TimerDisable(unsigned long ulBase, unsigned long ulTimer);
unsigned long TimerIntStatus(unsigned long ulBase, tBoolean bMasked);
void TimerIntClear(unsigned long ulBase, unsigned long ulIntFlags);
void TimerControlLevel(unsigned long ulBase, unsigned long ulTimer,
                       tBoolean bInvert);
void TimerMatchSet(unsigned long ulBase, unsigned long ulTimer,
                   unsigned long ulValue);
```

```
unsigned long TimerMatchGet(unsigned long ulBase, unsigned long ulTimer);
unsigned long TimerValueGet(unsigned long ulBase, unsigned long ulTimer);
void TimerControlEvent(unsigned long ulBase, unsigned long ulTimer,
                       unsigned long ulEvent);
void TimerControlStall(unsigned long ulBase, unsigned long ulTimer,
                       tBoolean bStall);
void TimerDMAEventSet(unsigned long ulBase, unsigned long ulDMAEvent);
unsigned long TimerDMAEventGet(unsigned long ulBase);
```

6．ADC：19 个

```
void ADCChannelEnable(unsigned long ulBase, unsigned long ulChannel);
void ADCChannelDisable(unsigned long ulBase, unsigned long ulChannel);
void ADCTimerConfig(unsigned long ulBase, unsigned long ulValue);
void ADCTimerEnable(unsigned long ulBase);
void ADCTimerDisable(unsigned long ulBase);
void ADCTimerReset(unsigned long ulBase);
void ADCEnable(unsigned long ulBase);
void ADCDisable(unsigned long ulBase);
unsigned long ADCTimerValueGet(unsigned long ulBase);
unsigned char ADCFIFOLvlGet(unsigned long ulBase, unsigned long ulChannel);
unsigned long ADCFIFORead(unsigned long ulBase, unsigned long ulChannel);
void ADCIntRegister(unsigned long ulBase, unsigned long ulChannel,
                    void (*pfnHandler)(void));
void ADCIntUnregister(unsigned long ulBase, unsigned long ulChannel);
void ADCIntEnable(unsigned long ulBase, unsigned long ulChannel,
                  unsigned long ulIntFlags);
void ADCIntDisable(unsigned long ulBase, unsigned long ulChannel,
                   unsigned long ulIntFlags);
unsigned long ADCIntStatus(unsigned long ulBase, unsigned long ulChannel);
void ADCIntClear(unsigned long ulBase, unsigned long ulChannel,
                 unsigned long ulIntFlags);
void ADCDMAEnable(unsigned long ulBase, unsigned long ulChannel);
void ADCDMADisable(unsigned long ulBase, unsigned long ulChannel);
```

7．Interrupt：15 个

```
void IntVTableBaseSet(unsigned long ulVtableBase);
tBoolean IntMasterEnable(void);
tBoolean IntMasterDisable(void);
void IntEnable(unsigned long ulInterrupt);
void IntDisable(unsigned long ulInterrupt);
void IntRegister(unsigned long ulInterrupt, void (*pfnHandler)(void));
void IntUnregister(unsigned long ulInterrupt);
unsigned long IntPriorityGroupingGet(void);
void IntPriorityGroupingSet(unsigned long ulBits);
long IntPriorityGet(unsigned long ulInterrupt);
void IntPrioritySet(unsigned long ulInterrupt, unsigned char ucPriority);
```

```
void IntPriorityMaskSet(unsigned long ulPriorityMask);
unsigned long IntPriorityMaskGet(void);
void IntPendSet(unsigned long ulInterrupt);
void IntPendClear(unsigned long ulInterrupt);
```

8．uDMA：24 个

```
void uDMAEnable(void);
void uDMADisable(void);
void uDMAControlBaseSet(void *pControlTable);
void *uDMAControlBaseGet(void);
void *uDMAControlAlternateBaseGet(void);
void uDMAChannelControlSet(unsigned long ulChannelStructIndex,
                           unsigned long ulControl);
void uDMAChannelAttributeEnable(unsigned long ulChannelNum,
                                unsigned long ulAttr);
void uDMAChannelAttributeDisable(unsigned long ulChannelNum,
                                 unsigned long ulAttr);
unsigned long uDMAChannelAttributeGet(unsigned long ulChannelNum);
void uDMAChannelTransferSet(unsigned long ulChannelStructIndex,
                            unsigned long ulMode, void *pvSrcAddr,
                            void *pvDstAddr, unsigned long ulTransferSize);
void uDMAChannelEnable(unsigned long ulChannelNum);
void uDMAChannelDisable(unsigned long ulChannelNum);
void uDMAChannelRequest(unsigned long ulChannelNum);
unsigned long uDMAChannelModeGet(unsigned long ulChannelStructIndex);
tBoolean uDMAChannelIsEnabled(unsigned long ulChannelNum);
void uDMAChannelAssign(unsigned long ulMapping);
void uDMAChannelScatterGatherSet(unsigned long ulChannelNum,
                                 unsigned ulTaskCount, void *pvTaskList,
                                 unsigned long ulIsPeriphSG);
void uDMAIntRegister(unsigned long ulIntChannel, void (*pfnHandler)(void));
void uDMAIntUnregister(unsigned long ulIntChannel);
unsigned long uDMAIntStatus(void);
void uDMAIntClear(unsigned long ulChanMask);
unsigned long uDMAErrorStatusGet(void);
void uDMAErrorStatusClear(void);
unsigned long uDMAChannelSizeGet(unsigned long ulChannelStructIndex);
```

9．PIN：15 个

```
void PinTypeGPIO(unsigned long ulPin,unsigned long ulPinMode,
                 tBoolean bOpenDrain);
void PinConfigSet(unsigned long ulPin, unsigned long ulPinStrength,
                  unsigned long ulPinType);
void PinConfigGet(unsigned long ulPin, unsigned long *pulPinStrength,
                  unsigned long *pulPinType);
void PinModeSet(unsigned long ulPin, unsigned long ulPinMode);
unsigned long PinModeGet(unsigned long ulPin);
```

```
void PinDirModeSet(unsigned long ulPin, unsigned long ulPinIO);
unsigned long PinDirModeGet(unsigned long ulPin);
void PinTypeUART(unsigned long ulPin,unsigned long ulPinMode);
void PinTypeSPI(unsigned long ulPin,unsigned long ulPinMode);
void PinTypeI2C(unsigned long ulPin,unsigned long ulPinMode);
void PinTypeI2S(unsigned long ulPin,unsigned long ulPinMode);
void PinTypeTimer(unsigned long ulPin,unsigned long ulPinMode);
void PinTypeADC(unsigned long ulPin,unsigned long ulPinMode);
void PinTypeCamera(unsigned long ulPin,unsigned long ulPinMode);
void PinTypeSDHost(unsigned long ulPin,unsigned long ulPinMode);
```

10．PRCM：43 个

```
void PRCMCC3200MCUInit(void);
void PRCMPeripheralClkEnable(unsigned long ulPeripheral,
                             unsigned long ulClkFlags);
void PRCMPeripheralClkDisable(unsigned long ulPeripheral,
                              unsigned long ulClkFlags);
unsigned long PRCMPeripheralClockGet(unsigned long ulPeripheral);
void PRCMPeripheralReset(unsigned long ulPeripheral);
void PRCMSleepEnter(void);
void PRCMDeepSleepEnter(void);
unsigned long PRCMSysResetCauseGet(void);
void PRCMHibernateIntervalSet(unsigned long long ullTicks);
void PRCMHibernateWakeupSourceEnable(unsigned long ulHIBWakupSrc);
void PRCMHibernateEnter(void);
void PRCMHibernateWakeupSourceDisable(unsigned long ulHIBWakupSrc);
unsigned longPRCMHibernateWakeupCauseGet(void);
void PRCMHibernateWakeUpGPIOSelect(unsigned long ulMultiGPIOBitMap,
                                   unsigned long ulType);
void PRCMLPDSIntervalSet(unsigned long ulTicks);
void PRCMLPDSWakeupSourceEnable(unsigned long ulLpdsWakeupSrc);
void PRCMLPDSWakeupSourceDisable(unsigned long ulLpdsWakeupSrc);
unsigned longPRCMLPDSWakeupCauseGet(void);
void PRCMLPDSWakeUpGPIOSelect(unsigned long ulGPIOPin,
                              unsigned long ulType);
void PRCMLPDSRestoreInfoSet(unsigned long ulRestoreSP,
                            unsigned long ulRestorePC);
void PRCMLPDSEnter(void);
void PRCMSOCReset(void);
void PRCMMCUReset(tBoolean bIncludeSubsystem);
tBoolean PRCMPeripheralStatusGet(unsigned long ulPeripheral);
void PRCMI2SClockFreqSet(unsigned long ulI2SClkFreq);
void PRCMSRAMRetentionEnable(unsigned long ulSramColSel,
                             unsigned long ulFlags);
void PRCMSRAMRetentionDisable(unsigned long ulSramColSel,
                              unsigned long ulFlags);
unsigned long PRCMSlowClkCtrGet(void);
```

```
void PRCMSlowClkCtrMatchSet(unsigned long long ullTicks);
unsigned long PRCMSlowClkCtrMatchGet(void);
void PRCMOCRRegisterWrite(unsigned char ucIndex, unsigned long ulRegValue);
unsigned long PRCMOCRRegisterRead(unsigned char ucIndex);
void PRCMIntRegister(void (*pfnHandler)(void));
void PRCMIntUnregister(void);
void PRCMIntEnable(unsigned long ulIntFlags);
void PRCMIntDisable(unsigned long ulIntFlags);
unsigned long PRCMIntStatus(void);
void PRCMRTCSet(unsigned long ulSecs, unsigned short usMsec);
void PRCMRTCGet(unsigned long *ulSecs, unsigned short *usMsec);
void PRCMRTCMatchSet(unsigned long ulSecs, unsigned short usMsec);
void PRCMRTCMatchGet(unsigned long *ulSecs, unsigned short *usMsec);
void PRCMRTCInUseSet(void);
tBoolean PRCMRTCInUseGet(void);
```

11．I2S：20 个

```
void I2SConfigSetExpClk(unsigned long ulBase, unsigned long ulI2SClk,
                        unsigned long ulBitClk, unsigned long ulConfig);
void I2SEnable(unsigned long ulBase, unsigned long ulMode);
void I2SDisable(unsigned long ulBase);
void I2SDataPut(unsigned long ulBase, unsigned long ulDataLine,
                unsigned long ulData);
long I2SDataPutNonBlocking(unsigned long ulBase, unsigned long ulDataLine,
                           unsigned long ulData);
void I2SDataGet(unsigned long ulBase, unsigned long ulDataLine,
                unsigned long *pulData);
long I2SDataGetNonBlocking(unsigned long ulBase, unsigned long ulDataLine,
                           unsigned long *pulData);
void I2SintRegister(unsigned long ulBase, void (*pfnHandler)(void));
void I2SIntUnregister(unsigned long ulBase);
void I2SIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
void I2SIntDisable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long I2SIntStatus(unsigned long ulBase);
void I2SIntClear(unsigned long ulBase, unsigned long ulIntFlags);
void I2STxFIFOEnable(unsigned long ulBase, unsigned long ulTxLevel,
                     unsigned long ulWordsPerTransfer);
void I2STxFIFODisable(unsigned long ulBase);
unsigned long I2STxFIFOStatusGet(unsigned long ulBase);
void I2SRxFIFOEnable(unsigned long ulBase, unsigned long ulRxLevel,
                     unsigned long ulWordsPerTransfer);
void I2SRxFIFODisable(unsigned long ulBase);
unsigned long I2SRxFIFOStatusGet(unsigned long ulBase);
void I2SSerializerConfig(unsigned long ulBase, unsigned long ulDataLine,
                         unsigned long ulSerMode, unsigned long ulInActState);
```

12．Camera：16 个

```
Void CameraReset (unsigned long ulBase);
Void CameraParamsConfig(unsigned long ulBase, unsigned long ulHSPol,
                        unsigned long ulVSPol, unsigned long ulFlags);
void CameraXClkConfig(unsigned long ulBase, unsigned long ulCamClkIn,
                      unsigned long ulXClk);
void CameraXClkSet(unsigned long ulBase, unsigned char bXClkFlags);
void CameraThresholdSet(unsigned long ulBase, unsigned long ulThreshold);
void CameraIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void CameraIntUnregister(unsigned long ulBase);
void CameraIntEnable (unsigned long ulBase, unsigned long ulIntFlags);
void CameraIntDisable (unsigned long ulBase, unsigned long ulIntFlags);
unsigned longCameraIntStatus(unsigned long ulBase);
voidCameraIntClear (unsigned long ulBase, unsigned long ulIntFlags);
voidCameraCaptureStart(unsigned long ulBase);
voidCameraCaptureStop(unsigned long ulBase, tBoolean bImmediate);
voidCameraBufferRead(unsigned long ulBase, unsigned long *pBuffer,
                     unsigned char ucSize);
voidCameraDMAEnable(unsigned long ulBase);
voidCameraDMADisable(unsigned long ulBase);
```

13．SDCARD：19 个

```
Void SDHostInit(unsigned long ulBase);
Void SDHostCmdReset(unsigned long ulBase);
long SDHostCmdSend(unsigned long ulBase, unsigned long ulCmd,
                   unsigned ulArg);
tBooleanSDHostDataNonBlockingWrite(unsigned long ulBase,
                                   unsigned long ulData);
void SDHostDataWrite(unsigned long ulBase, unsigned long ulData);
void SDHostDataRead (unsigned long ulBase, unsigned long *pulData);
tBooleanSDHostDataNonBlockingRead(unsigned long ulBase,
                                  unsigned long *pulData);
void SDHostIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void SDHostIntUnregister(unsigned long ulBase);
void SDHostIntEnable(unsigned long ulBase, unsigned long ulIntFlags);
void SDHostIntDisable(unsigned long ulBase, unsigned long ulIntFlags);
unsigned long SDHostIntStatus(unsigned long ulBase);
void SDHostIntClear(unsigned long ulBase, unsigned long ulIntFlags);
void SDHostCardErrorMaskSet(unsigned long ulBase, unsigned long ulErrMask);
unsigned long SDHostCardErrorMaskGet(unsigned long ulBase);
void SDHostSetExpClk(unsigned long ulBase, unsigned long ulSDHostClk,
                     unsigned long ulCardClk);
void SDHostRespGet(unsigned long ulBase, unsigned long ulRespnse[4]);
void SDHostBlockSizeSet(unsigned long ulBase, unsigned short ulBlkSize);
void SDHostBlockCountSet(unsigned long ulBase, unsigned short ulBlkCount);
```

14．Flash：14 个

```
Long FlashErase (unsigned long ulAddress);
Void FlashEraseNonBlocking(unsigned long ulAddress);
Long FlashMassErase(void);
Void FlashMassEraseNonBlocking(void);
tFlashProtection FlashProtectGet(unsigned long ulAddress);
long FlashProgram(unsigned long *pulData, unsigned long ulAddress,
                  unsigned long ulCount);
long FlashProgramNonBlocking(unsigned long *pulData,
                             unsigned long ulAddress,
                             unsigned long ulCount);
void FlashDisable(void);
void FlashIntRegister(void(*pfnHandler)(void));
void FlashIntUnregister(void);
void FlashIntEnable(unsigned long ulIntFlags);
void FlashIntDisable(unsigned long ulIntFlags);
unsigned long FlashIntStatus (tBoolean bMasked);
void FlashIntClear (unsigned long ulIntFlags);
```

15．SYSTICK：9 个

```
void SysTickEnable(void);
void SysTickDisable(void);
void SysTickIntRegister(void (*pfnHandler)(void));
void SysTickIntUnregister(void);
void SysTickIntEnable(void);
void SysTickIntDisable(void);
void SysTickPeriodSet(unsigned long ulPeriod);
unsigned long SysTickPeriodGet(void);
unsigned long SysTickValueGet(void);
```

16．WDT：15 个

```
void WatchdogEnable(unsigned long ulBase);
void WatchdogLock(unsigned long ulBase);
void WatchdogUnlock(unsigned long ulBase);
tBoolean WatchdogLockState(unsigned long ulBase);
void WatchdogReloadSet(unsigned long ulBase, unsigned long ulLoadVal);
unsigned long WatchdogReloadGet(unsigned long ulBase);
unsigned long WatchdogValueGet(unsigned long ulBase);
tBoolean WatchdogRunning(unsigned long ulBase);
void WatchdogIntRegister(unsigned long ulBase, void(*pfnHandler)(void));
void WatchdogIntUnregister(unsigned long ulBase);
void WatchdogIntEnable(unsigned long ulBase);
unsigned long WatchdogIntStatus(unsigned long ulBase, tBoolean bMasked);
void WatchdogIntClear(unsigned long ulBase);
void WatchdogStallEnable(unsigned long ulBase);
void WatchdogStallDisable(unsigned long ulBase);
```

附录 B

CC3200 网络驱动 API

1. DEVICE：6 个

```
_i16 sl_Start(const void *pIfHdl, _i8 *pDevName,
              const P_INIT_CALLBACK pInitCallBack);5.1(165)
_i16 sl_Stop(_u16 timeout); 5.1(169)
_i32 sl_DevSet(_u8 DeviceSetId, _u8 Option, _u8 ConfigLen,
               _u8 *pValues);
_i32 sl_DevGet(_u8 DeviceGetId, _u8 *pOption, _u8 *pConfigLen,
               _u8 *pValues); 5.1(166)
_i16 sl_EventMaskSet(_u8 EventClass , _u32 Mask);
_i16 sl_EventMaskGet(_u8 EventClass, _u32 *pMask);
```

2. WLAN：16 个

```
_i16 sl_WlanPolicySet(_u8 Type, const _u8 Policy, _u8 *pVal, _u8 ValLen);
_i16 sl_WlanPolicyGet(_u8 Type, _u8 Policy, _u8 *pVal, _u8 *pValLen);
_i16 sl_WlanSet(_u16 ConfigId, _u16 ConfigOpt,
                _u16 ConfigLen, _u8 *pValues);
_i16 sl_WlanGet(_u16 ConfigId, _u16 *pConfigOpt,
                _u16 *pConfigLen, _u8 *pValues);
_i16 sl_WlanConnect(_i8 *pName, _i16 NameLen, _u8 *pMacAddr,
                    SlSecParams_t *pSecParams,
                    SlSecParamsExt_t *pSecExtParams);
_i16 sl_WlanDisconnect(void);
_i16 sl_WlanSetMode(const _u8 mode);
_i16 sl_WlanSmartConfigStart(const _u32 groupIdBitmask, const _u8 cipher,
                             const _u8 publicKeyLen, const _u8 group1KeyLen,
                             const _u8 group2KeyLen, const _u8 *publicKey,
                             const _u8 *group1Key, const _u8 *group2Key);
_i16 sl_WlanSmartConfigStop(void);
_i16 sl_WlanGetNetworkList(_u8 Index, _u8 Count,
                           Sl_WlanNetworkEntry_t *pEntries);
_i16 sl_WlanProfileAdd(_i8 *pName, _i16 NameLen, _u8 *pMacAddr,
                       SlSecParams_t *pSecParams,
                       SlSecParamsExt_t *pSecExtParams,
                       _u32 Priority, _u32 Options);
_i16 sl_WlanProfileGet(_i16 Index, _i8 *pName, _i16 *pNameLen,
                       _u8 *pMacAddr, SlSecParams_t *pSecParams,
```

```
                        SlGetSecParamsExt_t *pSecExtParams, _u32 *pPriority);
_i16 sl_WlanProfileDel(_i16 Index);
_i16 sl_WlanRxStatStart(void);
_i16 sl_WlanRxStatStop(void);
_i16 sl_WlanRxStatGet(SlGetRxStatResponse_t *pRxStat, _u32 Flags);
```

3．Socket：15 个

```
_i16 sl_Socket(_i16 Domain, _i16 Type, _i16 Protocol);
_i16 sl_Bind(_i16 sd, const SlSockAddr_t *addr, _i16 addrlen);
_i16 sl_Listen(_i16 sd, _i16 backlog);
_i16 sl_SetSockOpt(_i16 sd, _i16 level, _i16 optname,
                   const void *optval, SlSocklen_t optlen);
_i16 sl_GetSockOpt(_i16 sd, _i16 level, _i16 optname,
                   void *optval, SlSocklen_t *optlen);
_i16 sl_Accept(_i16 sd, SlSockAddr_t *addr, SlSocklen_t *addrlen);
_i16 sl_Connect(_i16 sd, const SlSockAddr_t *addr, _i16 addrlen);
_i16 sl_Recv(_i16 sd, void *buf, _i16 Len, _i16 flags);
_i16 sl_Send(_i16 sd, const void *buf, _i16 Len, _i16 flags);
_i16 sl_Close(_i16 sd); 6.1(192)6.1(195)6.2(201)6.2(203)
_i16 sl_RecvFrom(_i16 sd, void *buf, _i16 Len, _i16 flags,
                 SlSockAddr_t *from, SlSocklen_t *fromlen);
_i16 sl_SendTo(_i16 sd, const void *buf, _i16 Len, _i16 flags,
               const SlSockAddr_t *to, SlSocklen_t tolen);
_i16 sl_Select(_i16 nfds, SlFdSet_t *readsds, SlFdSet_t *writesds,
               SlFdSet_t *exceptsds, struct SlTimeval_t *timeout);
_u32 sl_Htonl(_u32 val);
_u16 sl_Htons(_u16 val);
```

4．NETAPP：10 个

```
_i16 sl_NetAppPingStart(SlPingStartCommand_t *pPingParams,
                        _u8 family, SlPingReport_t *pReport,
                        const P_SL_DEV_PING_CALLBACK pPingCallback);
_i16 sl_NetAppDnsGetHostByName(_i8 *hostname, _u16 usNameLen,
                               _u32 *out_ip_addr, _u8 family);
_i32 sl_NetAppDnsGetHostByService(_i8 *pServiceName, _u8 ServiceLen,
                                  _u8 Family, _u32 pAddr[], _u32 *pPort,
                                  _u16 *pTextLen, _i8 *pText);
_i16 sl_NetAppMDNSRegisterService(const _i8 *pServiceName,
                            _u8 ServiceNameLen, const _i8 *pText,
                            _u8 TextLen, _u16 Port, _u32 TTL, _u32 Options);
_i16 sl_NetAppMDNSUnRegisterService(const _i8 *pServiceName,
                                    _u8 ServiceNameLen);
_i16 sl_NetAppGetServiceList(_u8 IndexOffest, _u8 MaxServiceCount,
                             _u8 Flags, _i8 *pBuffer, _u32 RxBufferLength);
_i16 sl_NetAppStart(_u32 AppBitMap);
_i16 sl_NetAppStop(_u32 AppBitMap);
_i32 sl_NetAppSet(_u8 AppId, _u8 Option, _u8 OptionLen, _u8 *pOptionValue);
```

```
_i32 sl_NetAppGet(_u8 AppId, _u8 Option, _u8 *pOptionLen, _u8 *pOptionValue);
```

5．NETCFG：2 个

```
_i32 sl_NetCfgSet(_u8 ConfigId, _u8 ConfigOpt, _u8 ConfigLen, _u8 *pValues);
_i32 sl_NetCfgGet(_u8 ConfigId, _u8 *pConfigOpt, _u8 *pConfigLen,
                  _u8 *pValues);
```

6．FileSystem：6 个

```
_i32 sl_FsOpen(_u8 *pFileName, _u32 AccessModeAndMaxSize,
               _u32 *pToken, _i32 *pFileHandle);
_i32 sl_FsRead(_i32 FileHdl, _u32 Offset, _u8 *pData, _u32 Len);
_i32 sl_FsWrite(_i32 FileHdl, _u32 Offset, _u8 *pData, _u32 Len);
_i16 sl_FsGetInfo(_u8 *pFileName, _u32 Token, SlFsFileInfo_t *pFsFileInfo);
_i16 sl_FsDel(_u8 *pFileName,_u32 Token);
_i16 sl_FsClose(_i32 FileHdl, _u8 *pCeritificateFileName,
                _u8 *pSignature, _u32 SignatureLen);
```

7．UserEvent：4 个

```
void sl_WlanEvtHdlr(SlWlanEvent_t * pSlWlanEvent);
void sl_NetAppEvtHdlr(SlNetAppEvent_t * pSlNetApp);
void sl_SockEvtHdlr(SlSockEvent_t * pSlSockEvent);
voidsl_HttpServerCallback(SlHttpServerEvent_t * pSlHttpServerEvent,
                          SlHttpServerResponse_t * pSlHttpServerResponse);
```

附录 C

CC3200 引脚复用功能

引脚	名　　称	模式值	信号名称	信号说明	信号方向
1	GPIO10	0	GPIO10	GPIO	输入/输出
		1	I2C_SCL	I2C 时钟	输出（开漏）
		3	GT_PWM06	PWM O/P	输出
		6	SDCARD_CLK	SD 卡时钟	输出
		7	UART1_TX	UART 发送数据	输出
		12	GT_CCP01	定时器捕捉	输入
2	GPIO11	0	GPIO11	GPIO	输入/输出
		1	I2C_SDA	I2C 数据	输入/输出（开漏）
		3	GT_PWM07	PWM O/P	输出
		4	pXCLK(XVCLK)	摄像头时钟	输出
		6	SDCARD_CMD	SD 卡命令	输入/输出
		7	UART1_RX	UART 接收数据	输入
		12	GT_CCP02	定时器捕捉	输入
		13	McAFSX	I2S 帧同步	输出
3	GPIO12	0	GPIO12	GPIO	输入/输出
		3	McACLK	I2S 时钟	输出
		4	pVS(VSYNC)	摄像头垂直同步	输入
		5	I2C_SCL	I2C 时钟	输入/输出（开漏）
		7	UART0_TX	UART 发送数据	输出
		12	GT_CCP03	定时器捕捉	输入
4	GPIO13	0	GPIO13	GPIO	输入/输出
		4	pHS(HSYNC)	摄像头水平同步	输入
		5	I2C_SDA	I2C 数据	输入/输出（开漏）
		7	UART0_RX	UART 接收数据	输入
		12	GT_CCP04	定时器捕捉	输入
5	GPIO14	0	GPIO14	GPIO	输入/输出
		4	pDATA8(CAM_D4)	摄像头数据位 4	输入
		5	I2C_SCL	I2C 时钟	输入/输出（开漏）
		7	GSPI_CLK	通用 SPI 时钟	输入/输出
		12	GT_CCP05	定时器捕捉	输入
6	GPIO15	0	GPIO15	GPIO	输入/输出
		4	pDATA9(CAM_D5)	摄像头数据位 5	输入
		5	I2C_SDA	I2C 数据	输入/输出（开漏）
		7	GSPI_MISO	通用 SPI MISO	输入/输出
		13	GT_CCP06	定时器捕捉	输入

续表

引脚	名　　称	模式值	信号名称	信号说明	信号方向
7	GPIO16	0	GPIO16	GPIO	输入/输出
		4	pDATA10(CAM_D6)	摄像头数据位 6	输入
		5	UART1_TX	UART 发送数据	输出
		7	GSPI_MOSI	通用 SPI MOSI	输入/输出
		13	GT_CCP07	定时器捕捉	输入
8	GPIO17	0	GPIO17	GPIO	输入/输出
		4	pDATA11(CAM_D7)	摄像头数据位 7	输入
		5	UART1_RX	UART 接收数据	输入
		7	GSPI_CS	通用 SPI 片选	输入/输出
9	VDD_DIG1	—	VDD_DIG1	内部数字电源	
10	VIN_IO1	—	VIN_IO1	芯片电源	
11	FLASH_SPI_CLK	—	FLASH_SPI_CLK	串行闪存 SPI 时钟	输出
12	FLASH_SPI_DOUT	—	FLASH_SPI_DOUT	串行闪存 SPI 数据输出	输出
13	FLASH_SPI_DIN	—	FLASH_SPI_DIN	串行闪存 SPI 数据输入	输入
14	FLASH_SPI_CS	—	FLASH_SPI_CS	串行闪存 SPI 片选	输出
15	GPIO22	0	GPIO22	GPIO	输入/输出
		5	GT_CCP04	定时器捕捉	输入
		7	McAFSX	I2S 帧同步	输出
16	TDI	0	GPIO23	GPIO	输入/输出
		1	TDI	JTAG TDI	输入
		2	UART1_TX	UART 发送数据	输出
		9	I2C_SCL	I2C 时钟	输入/输出（开漏）
17	TDO	0	GPIO24	GPIO	输入/输出
		1	TDO	JTAG TDO	输出
		2	UART1_RX	UART 接收数据	输入
		4	GT_CCP06	定时器捕捉	输入
		5	GT_PWM00	PWM O/P	输出
		6	McAFSX	I2S 帧同步	输出
		9	I2C_SDA	I2C 数据	输入/输出（开漏）
18	GPIO28	0	GPIO28	GPIO	输入/输出
19	TCK	1	TCK	JTAG/SWD TCK	输入
		8	GT_PWM03	PWM O/P	输出
20	TMS	0	GPIO29	GPIO	输入/输出
		1	TMS	JTAG/SWD TMS	输入/输出
21	SOP2		SOP2	SOP2	输入
			TCXO_EN	40MHz TCXO 允许	输出
		0	GPIO25	GPIO	输出
		2	McAFSX	I2S 帧同步	输出
		9	GT_PWM02	PWM O/P	输出
22	WLAN_XTAL_N		WLAN_XTAL_N	40 MHz XTAL 下拉	
23	WLAN_XTAL_P		WLAN_XTAL_P	40 MHz XTAL 时钟输入	
24	VDD_PLL		VDD_PLL	内部模拟电源	
25	LDO_IN2		LDO_IN2	模拟视频电源	
26	NC		NC	保留	

续表

引脚	名　　称	模式值	信号名称	信号说明	信号方向
27	NC		NC	保留	
28	NC		NC	保留	
29	ANTSEL1		ANTSEL1	天线选择控制 1	输出
30	ANTSEL2		ANTSEL2	天线选择控制 2	输出
31	RF_BG		RF_BG	射频 BG 带	
32	nRESET		nRESET	芯片复位	
33	VDD_PA_IN		VDD_PA_IN	功放电源	
34	SOP1		SOP1	SOP1	
35	SOP0		SOP0	SOP0	
36	LDO_IN1		LDO_IN1	模拟视频电源	
37	VIN_DCDC_ANA		VIN_DCDC_ANA	模拟 DCDC 输入	
38	DCDC_ANA_SW		DCDC_ANA_SW	模拟 DCDC 开关	
39	VIN_DCDC_PA		VIN_DCDC_PA	功放 DCDC 输入	
40	DCDC_PA_SW_P		DCDC_PA_SW_P	功放 DCDC 开关	
41	DCDC_PA_SW_N		DCDC_PA_SW_N	功放 DCDC 开关	
42	DCDC_PA_OUT		DCDC_PA_OUT	功放 DCDC 输出	
43	DCDC_DIG_SW		DCDC_DIG_SW	数字 DCDC 开关	
44	VIN_DCDC_DIG		VIN_DCDC_DIG	数字 DCDC 输入	
45	DCDC_ANA2_SW_P		DCDC_ANA2_SW_P	模拟 DCDC 开关	
		0	GPIO31	GPIO	输入/输出
		2	UART1_RX	UART 接收数据	输入
		6	McAXR0	I2S 数据 0	输入/输出
		7	GSPI_CLK	通用 SPI 时钟	输入/输出
		9	UART0_RX	UART 接收数据	输入
		12	McAFSX	I2S 帧同步	输出
46	DCDC_ANA2_SW_N		DCDC_ANA2_SW_N	模拟 DCDC 开关	
47	VDD_ANN2		VDD_ANN2	模拟电源输出	
48	VDD_ANA1		VDD_ANA1	模拟电源输出	
49	VDD_RAM		VDD_RAM	SRAM LDO 输出	
50	GPIO0	0	GPIO0	GPIO	输入/输出
		3	UART0_RTS	UART 请求发送	输出
		4	McAXR0	I2S 数据 0	输入/输出
		6	McAXR1	I2S 数据 1	输入/输出
		7	GT_CCP00	定时器捕捉	输入
		9	GSPA_CS	通用 SPI 片选	输入/输出
		10	UART1_RTS	UART 请求发送	输出
		12	UART0_CTS	UART 清除发送	输入
51	RTC_XTAL_P		RTC_XTAL_P	32KHz XTAL	
52	RTC_XTAL_N		RTC_XTAL_N	32KHz XTAL	
		0	GPIO32	GPIO	输入/输出
		2	McACLK	I2S 时钟	输出
		4	McAXR0	I2S 数据 0	输出
		6	UART0_RTS	UART 请求发送	输出
		8	GSPI_MOSI	通用 SPI MOSI	输入/输出

续表

引脚	名　称	模式值	信号名称	信号说明	信号方向
53	GPIO30	0	GPIO30	GPIO	输入/输出
		2	McACLK	I2S 时钟	输出
		3	McAFSX	I2S 帧同步	输出
		4	GT_CCP05	定时器捕捉	输入
		7	GSPI_MISO	通用 SPI MISO	输入/输出
		9	UART0_TX	UART 发送数据	输出
54	VIN_IO2		VIN_IO2	芯片电源	
55	GPIO1	0	GPIO1	GPIO	输入/输出
		3	UART0_TX	UART 发送数据	输出
		4	pCLK(PIXCLK)	摄像头像素时钟	输入
		6	UART1_TX	UART 发送数据	输出
		7	GT_CCP01	定时器捕捉	输入
56	VDD_DIG2		VDD_DIG2	内部数字电源	
57	GPIO2		ADC_CH0	ADC 通道 0（最大 1.4 V）	输入
		0	GPIO2	GPIO	输入/输出
		3	UART0_RX	UART 接收数据	输入
		6	UART1_RX	UART 接收数据	输入
		7	GT_CCP02	定时器捕捉	输入
58	GPIO3		ADC_CH1	ADC 通道 1（最大 1.4 V）	输入
		0	GPIO3	GPIO	输入/输出
		4	pDATA7(CAM_D3)	摄像头数据位 3	输入
		6	UART1_TX	UART 发送数据	输出
59	GPIO4		ADC_CH2	ADC 通道 2（最大 1.4 V）	输入
		0	GPIO4	GPIO	输入/输出
		4	pDATA6(CAM_D2)	摄像头数据位 2	输入
		6	UART1_RX	UART 接收数据	输入
60	GPIO5		ADC_CH3	ADC 通道 3（最大 1.4 V）	输入
		0	GPIO5	GPIO	输入/输出
		4	pDATA5(CAM_D1)	摄像头数据位 1	输入
		6	McAXR1	I2S 数据 1	输入/输出
		7	GT_CCP05	定时器捕捉	输入
61	GPIO6	0	GPIO6	GPIO	输入/输出
		3	UART1_CTS	UART 清除发送	输入
		4	pDATA4(CAM_D0)	摄像头数据位 0	输入
		5	UART0_RTS	UART 请求发送	输出
		6	UART0_CTS	UART 清除发送	输入
		7	GT_CCP06	定时器捕捉	输入
62	GPIO7	0	GPIO7	GPIO	输入/输出
		3	UART1_RTS	UART 请求发送	输出
		10	UART0_RTS	UART 请求发送	输出
		11	UART0_TX	UART 发送数据	输出
		13	McACLK	I2S 时钟	输出
63	GPIO8	0	GPIO8	GPIO	输入/输出
		6	SDCARD_IRQ	SD 卡中断	输入
		7	McAFSX	I2S 帧同步	输出
		12	GT_CCP06	定时器捕捉	输入
64	GPIO9	0	GPIO9	GPIO	输入/输出
		3	GT_PWM05	PWM O/P	输出
		6	SDCARD_DATA	SD 卡数据	输入/输出
		7	McAXR0	I2S 数据 0	输入/输出
		12	GT_CCP00	定时器捕捉	输入

附录 D

CC3200 外设引脚分配

表 D.1　GPIO 引脚

引脚	名　　称	模式值	信号名称	信号说明	信号方向
50	GPIO0	0	GPIO0（GPIOA0.0）	GPIO	输入/输出
55	GPIO1	0	GPIO1（GPIOA0.1）	GPIO	输入/输出
57	GPIO2	0	GPIO2（GPIOA0.2）	GPIO	输入/输出
58	GPIO3	0	GPIO3（GPIOA0.3）	GPIO	输入/输出
59	GPIO4	0	GPIO4（GPIOA0.4）	GPIO	输入/输出
60	GPIO5	0	GPIO5（GPIOA0.5）	GPIO	输入/输出
61	GPIO6	0	GPIO6（GPIOA0.6）	GPIO	输入/输出
62	GPIO7	0	GPIO7（GPIOA0.7）	GPIO	输入/输出
63	GPIO8	0	GPIO8（GPIOA1.0）	GPIO	输入/输出
64	GPIO9	0	GPIO9（GPIOA1.1）	GPIO	输入/输出
1	GPIO10	0	GPIO10（GPIOA1.2）	GPIO	输入/输出
2	GPIO11	0	GPIO11（GPIOA1.3）	GPIO	输入/输出
3	GPIO12	0	GPIO12（GPIOA1.4）	GPIO	输入/输出
4	GPIO13	0	GPIO13（GPIOA1.5）	GPIO	输入/输出
5	GPIO14	0	GPIO14（GPIOA1.6）	GPIO	输入/输出
6	GPIO15	0	GPIO15（GPIOA1.7）	GPIO	输入/输出
7	GPIO16	0	GPIO16（GPIOA2.0）	GPIO	输入/输出
8	GPIO17	0	GPIO17（GPIOA2.1）	GPIO	输入/输出
15	GPIO22	0	GPIO22（GPIOA2.6）	GPIO	输入/输出
16	TDI	0	GPIO23（GPIOA2.7）	GPIO	输入/输出
17	TDO	0	GPIO24（GPIOA3.0）	GPIO	输入/输出
21	SOP2	0	GPIO25（GPIOA3.1）	GPIO	输出
18	GPIO28	0	GPIO28（GPIOA3.4）	GPIO	输入/输出
20	TMS	0	GPIO29（GPIOA3.5）	GPIO	输入/输出
53	GPIO30	0	GPIO30（GPIOA3.6）	GPIO	输入/输出
45	DCDC_ANA2_SW_P	0	GPIO31（GPIOA3.7）	GPIO	输入/输出
52	RTC_XTAL_N	0	GPIO32（GPIOA4.0）	GPIO	输入/输出

表 D.2　UART 引脚

引脚	别　　名	模式值	信号名称	信号描述	信号方向
3	GPIO12	7	UART0_TX	UART 发送数据	输出
53	GPIO30	9	UART0_TX	UART 发送数据	输出
55	GPIO1	3	UART0_TX（默认）	UART 发送数据	输出
62	GPIO7	11	UART0_TX	UART 发送数据	输出

续表

引脚	别　名	模式值	信号名称	信号描述	信号方向
4	GPIO13	7	UART0_RX	UART 接收数据	输入
45	DCDC_ANA2_SW_P	9	UART0_RX	UART 接收数据	输入
57	GPIO2	3	UART0_RX（默认）	UART 接收数据	输入
50	GPIO0	3	UART0_RTS	UART 请求发送	输出
52	RTC_XTAL_N	6	UART0_RTS	UART 请求发送	输出
61	GPIO6	5	UART0_RTS	UART 请求发送	输出
62	GPIO7	10	UART0_RTS	UART 请求发送	输出
50	GPIO0	12	UART0_CTS	UART 清除发送	输入
61	GPIO6	6	UART0_CTS	UART 清除发送	输入
1	GPIO10	7	UART1_TX	UART 发送数据	输出
7	GPIO16	5	UART1_TX	UART 发送数据	输出
16	TDI	2	UART1_TX（默认）	UART 发送数据	输出
55	GPIO1	6	UART1_TX	UART 发送数据	输出
58	GPIO3	6	UART1_TX	UART 发送数据	输出
2	GPIO11	7	UART1_RX	UART 接收数据	输入
8	GPIO17	5	UART1_RX	UART 接收数据	输入
17	TDO	2	UART1_RX（默认）	UART 接收数据	输入
45	DCDC_ANA2_SW_P	2	UART1_RX	UART 接收数据	输入
57	GPIO2	6	UART1_RX	UART 接收数据	输入
59	GPIO4	6	UART1_RX	UART 接收数据	输入
50	GPIO0	10	UART1_RTS	UART 请求发送	输出
62	GPIO7	3	UART1_RTS	UART 请求发送	输出
61	GPIO6	3	UART1_CTS	UART 清除发送	输入

注：使用其他引脚实现默认功能时，必须把默认引脚配置为其他模式。

表 D.3　GSPI 引脚

引脚	别　名	模式值	信号名称	信号描述	信号方向
5	GPIO14	7	GSPI_CLK	通用 SPI 时钟	输入/输出
45	DCDC_ANA2_SW_P	7	GSPI_CLK	通用 SPI 时钟	输入/输出
6	GPIO15	7	GSPI_MISO	通用 SPI MISO	输入/输出
53	GPIO30	7	GSPI_MISO	通用 SPI MISO	输入/输出
7	GPIO16	7	GSPI_MOSI	通用 SPI MOSI	输入/输出
52	RTC_XTAL_N	7	GSPI_MOSI	通用 SPI MOSI	输入/输出
8	GPIO17	7	GSPI_CS	通用 SPI 片选	输入/输出
50	GPIO0	7	GSPI_CS	通用 SPI 片选	输入/输出

表 D.4　I2C 引脚

引脚	别　名	模式值	信号名称	信号描述	信号方向
1	GPIO10	1	I2C_SCL（默认）	I2C 时钟	输出（开漏）
3	GPIO12	5	I2C_SCL	I2C 时钟	输入/输出（开漏）
5	GPIO14	5	I2C_SCL	I2C 时钟	输入/输出（开漏）
16	TDI	9	I2C_SCL	I2C 时钟	输入/输出（开漏）
2	GPIO11	1	I2C_SDA（默认）	I2C 数据	输入/输出（开漏）
4	GPIO13	5	I2C_SDA	I2C 数据	输入/输出（开漏）
6	GPIO15	5	I2C_SDA	I2C 数据	输入/输出（开漏）
17	TDO	9	I2C_SDA	I2C 数据	输入/输出（开漏）

注：使用其他引脚实现默认功能时，必须把默认引脚配置为其他模式。

表 D.5　McASP 引脚

引脚	别　　名	模式值	信号名称	信号描述	信号方向
3	GPIO12	3	McACLK	I2S 时钟	输出
52	RTC_XTAL_N	2	McACLK	I2S 时钟	输出
53	GPIO30	2	McACLK	I2S 时钟	输出
62	GPIO7	13	McACLK	I2S 时钟	输出
2	GPIO11	13	McAFSX	I2S 帧同步	输出
15	GPIO22	7	McAFSX	I2S 帧同步	输出
17	TDO	6	McAFSX	I2S 帧同步	输出
21	SOP2	2	McAFSX	I2S 帧同步	输出
45	DCDC_ANA2_SW_P	12	McAFSX	I2S 帧同步	输出
53	GPIO30	3	McAFSX	I2S 帧同步	输出
63	GPIO8	7	McAFSX	I2S 帧同步	输出
45	DCDC_ANA2_SW_P	6	McAXR0	I2S 数据 0	输入/输出
50	GPIO0	6	McAXR0	I2S 数据 0	输入/输出
52	RTC_XTAL_N	4	McAXR0	I2S 数据 0	输出
64	GPIO9	7	McAXR0	I2S 数据 0	输入/输出
50	GPIO0	6	McAXR1	I2S 数据 1	输入/输出
60	GPIO5	6	McAXR1	I2S 数据 1	输入/输出

表 D.6　GPT 引脚

引脚	别　　名	模式值	信号名称	信号描述	信号方向
17	TDO	5	GT_PWM00	PWM O/P	输出
21	SOP2	9	GT_PWM02	PWM O/P	输出
19	TCK	8	GT_PWM03	PWM O/P	输出
64	GPIO9	3	GT_PWM05	PWM O/P	输出
1	GPIO10	3	GT_PWM06	PWM O/P	输出
2	GPIO11	3	GT_PWM07	PWM O/P	输出
50	GPIO0	7	GT_CCP00	定时器捕捉	输入
64	GPIO9	12	GT_CCP00	定时器捕捉	输入
1	GPIO10	12	GT_CCP01	定时器捕捉	输入
55	GPIO1	7	GT_CCP01	定时器捕捉	输入
2	GPIO11	12	GT_CCP02	定时器捕捉	输入
57	GPIO2	7	GT_CCP02	定时器捕捉	输入
3	GPIO12	12	GT_CCP03	定时器捕捉	输入
4	GPIO13	12	GT_CCP04	定时器捕捉	输入
15	GPIO22	5	GT_CCP04	定时器捕捉	输入
5	GPIO14	12	GT_CCP05	定时器捕捉	输入
53	GPIO30	4	GT_CCP05	定时器捕捉	输入
60	GPIO5	7	GT_CCP05	定时器捕捉	输入
6	GPIO15	13	GT_CCP06	定时器捕捉	输入
17	TDO	4	GT_CCP06	定时器捕捉	输入
61	GPIO6	7	GT_CCP06	定时器捕捉	输入
63	GPIO8	12	GT_CCP06	定时器捕捉	输入
7	GPIO16	13	GT_CCP07	定时器捕捉	输入

表 D.7 ADC 引脚

引脚	别 名	模式值	信号名称	信号描述	信号方向
57	GPIO2		ADC_CH0	ADC 通道 0（最大 1.4V）	输入
58	GPIO3		ADC_CH1	ADC 通道 1（最大 1.4V）	输入
59	GPIO4		ADC_CH2	ADC 通道 2（最大 1.4V）	输入
60	GPIO5		ADC_CH3	ADC 通道 3（最大 1.4V）	输入

表 D.8 SDCARD 引脚

引脚	别 名	模式值	信号名称	信号描述	信号方向
1	GPIO10	6	SDCARD_CLK	SD 卡时钟	输出
2	GPIO11	6	SDCARD_CMD	SD 卡命令	输入/输出
63	GPIO8	6	SDCARD_IRQ	SD 卡中断	输入
64	GPIO9	6	SDCARD_DATA	SD 卡数据	输入/输出

表 D.9 摄像头

引脚	别 名	模式值	信号名称	信号描述	信号方向
2	GPIO11	4	pXCLK(XVCLK)	摄像头时钟	输出
3	GPIO12	4	pVS(VSYNC)	摄像头垂直同步	输入
4	GPIO13	4	pHS(HSYNC)	摄像头水平同步	输入
55	GPIO1	4	pCLK(PIXCLK)	摄像头像素时钟	输入
61	GPIO6	4	pDATA4(CAM_D0)	摄像头数据位 0	输入
60	GPIO5	4	pDATA5(CAM_D1)	摄像头数据位 1	输入
59	GPIO4	4	pDATA6(CAM_D2)	摄像头数据位 2	输入
58	GPIO3	4	pDATA7(CAM_D3)	摄像头数据位 3	输入
5	GPIO14	4	pDATA8(CAM_D4)	摄像头数据位 4	输入
6	GPIO15	4	pDATA9(CAM_D5)	摄像头数据位 5	输入
7	GPIO16	4	pDATA10(CAM_D6)	摄像头数据位 6	输入
8	GPIO17	4	pDATA11(CAM_D7)	摄像头数据位 7	输入

表 D.10 JTAG 引脚

引脚	别 名	模式值	信号名称	信号描述	信号方向
16	TDI	1	TDI	JTAG TDI	输入
17	TDO	1	TDO	JTAG TDO	输出
19	TCK	1	TCK	JTAG/SWD TCK	输入
20	TMS	1	TMS	JTAG/SWD TMS	输入/输出

附录 E

CC3200 LaunchPad 简介

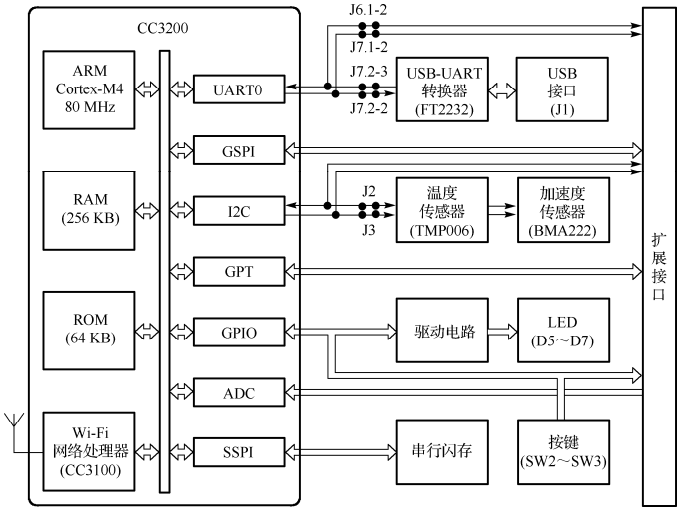


图 E.1　CC3200 LaunchPad 功能框图

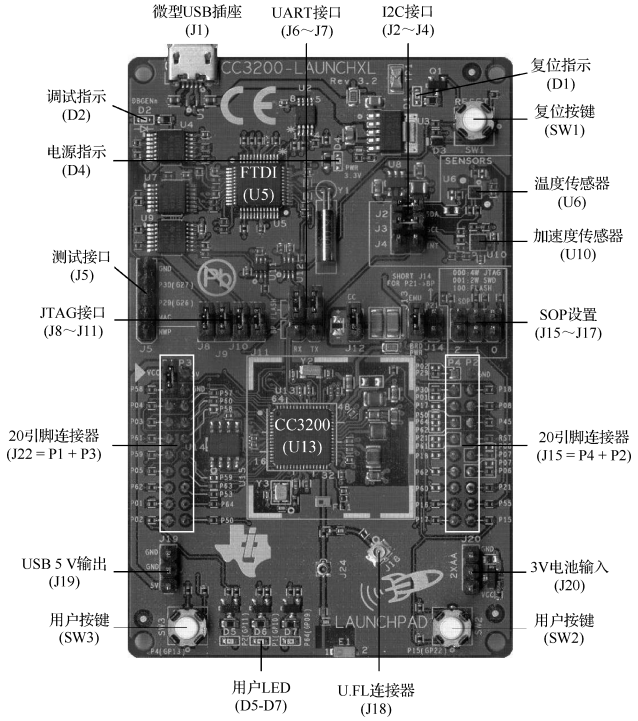


图 E.2　CC3200 LaunchPad 实物图

表 E.1 CC3200 LaunchPad 连接关系

部件	连接	功能	CC3200 引脚	说明
微型 USB 插座	J1	—	—	供电，调试，UART 通信
I2C 接口 温度传感器 加速度传感器	J2	SDA	2/GPIO11	I2C_SDA/P1.10/D5
	J3	SCL	1/GPIO10	I2C_SCL/P1.9/D6/UART1_TX
	J4	INT	4/GPIO13	加速度传感器中断/P1.3/SW3
测试接口	J5.1	NWP	62/GPIO7	P1.8/UART0_TX
	J5.2	MAC	60/GPIO5	P3.4
	J5.3	P29(G26)	29/GPIO26	P4.1
	J5.4	P30(G27)	30/GPIO27	P4.2
	J5.5	GND	—	GND
UART 接口 P3.3/P2.9 FTDI	J6.1-2	RX	57/GPIO2	P3.3
	J6.3-2	RX	57/GPIO2	USB-UART Flash 编程
	J7.1-2	TX	55/GPIO1	P2.9
	J7.3-2	TX	55/GPIO1	USB-UART FLASH 编程
JTAG 接口	J8	TCK	19/TCK	JTAG_TCK/SWD_CLK
	J9	TMS	20/TMS	JTAG_TMS/SWD_DIO
	J10	TDI	16/TDI	JTAG_TDI/P4.9/SCL
	J11	TDO	17/TDO	JTAG_TDO/P4.10/SDA
	J12	电流测量	—	短路：正常工作 开路：测量 CC3200 电流
	J13	电源选择	—	短路：USB 5 V 输入 开路：3 V 电池输入
	J14	—	21/SOP2	P2.8
SOP 设置	J15	SOP2	21/SOP2	开路-0，短路-1
	J16	SOP1	34/SOP1	开路-0，短路-1
	J17	SOP0	35/SOP0	开路-0，短路-1
U.FL 连接器	J18	—	—	射频测试
USB 5V 输出	J19.1	5 V	—	USB 5 V 输出
	J19.2-3	GND	—	GND
3V 电池输入	J20.1	2xAA	—	3 V 电池输入
	J20.2-3	GND	—	GND
20 引脚连接器 J22.P1	P1.1	VCC	—	3.3 V
	P1.2	P58	58/GPIO3	ADC_CH1/CAM_D3
	P1.3	P04	4/GPIO13	UART0_RX/INT/SW3/HSYNC
	P1.4	P03	3/GPIO12	UART0_TX/VSYNC
	P1.5	P61	61/GPIO6	CAM_D0
	P1.6	P59	59/GPIO4	ADC_CH2/CAM_D2
	P1.7	P05	5/GPIO14	GSPI_CLK/CAM_D4
	P1.8	P62	62/GPIO7	J5.1
	P1.9	P01	1/GPIO10	SCL/D6
	P1.10	P02	2/GPIO11	SDA/D5/XVCLK
20 引脚连接器 J23.P2	P2.1	GND	—	
	P2.2	P18	18/GPIO28	
	P2.3	P08	8/GPIO17	GSPI_CS/CAM_D7
	P2.4	P45	45/GPIO31	UART0_RX
	P2.5	RST	—	

续表

部件	连接	功能	CC3200 引脚	说明
20 引脚连接器 J23.P2	P2.6	P07 P06	7/GPIO16 (6/GPIO15)	GSPI_MOSI/CAM_D6
	P2.7	P06 P07	6/GPIO15 (7/GPIO16)	GSPI_MISO/CAM_D5
	P2.8	P21	21/SOP2	J14.1
	P2.9	P55	55/GPIO1	J7.1/PIXCLK
	P2.10	P15	15/GPIO22	SW2
20 引脚连接器 J22.P3	P3.1	5 V	—	
	P3.2	GND	—	
	P3.3	P57	57/GPIO2	ADC_CH0/J6.1
	P3.4	P60	60/GPIO5	ADC_CH3/J5.2/CAM_D1
	P3.5	P58	(58/GPIO3)	
	P3.6	P59	(59/GPIO4)	
	P3.7	P63	63/GPIO8	I2S_FSYNC
	P3.8	P53	53/GPIO30	I2S_CLK
	P3.9	P64	64/GPIO9	I2S_DOUT/D7
	P3.10	P50	50/GPIO0	I2S_DIN
20 引脚连接器 J23.P4	P4.1	P02 P29	(2/GPIO11) 29/GPIO26	(GT_PWM07) J5.3
	P4.2	P01 P30	(1/GPIO10) 30/GPIO27	(GT_PWM06) J5.4
	P4.3	P17	(17/GPIO24)	(GT_PWM00)
	P4.4	P50 P62 P64	(50/GPIO0) (62/GPIO7) (64/GPIO9)	(GT_PWM05)
	P4.5	P21 P61	(21/SOP2) (61/GPIO6)	(GT_CCP06)
	P4.6	P18	(18/GPIO28)	
	P4.7	P62	(62/GPIO7)	
	P4.8	P60	(60/GPIO5)	
	P4.9	P16	16/GPIO23	TDI/SCL
	P4.10	P17	17/GPIO24	TDO/SDA
按键		SW1	32/nRESET	复位按键
		SW2	15/GPIO22	用户按键/P2.10
		SW3	4/GPIO13	用户按键/INT/P1.3
LED		D1（黄）	—	复位指示
		D2（绿）	—	调试指示
		D4（红）	—	电源指示
		D5（绿）	2/GPIO11	用户 LED/SDA/P1.10
		D6（黄）	1/GPIO10	用户 LED/SCL/P1.9
		D7（红）	64/GPIO9	用户 LED/P3.9

附录 F

CC3200 异常和中断

中断号（矢量号）	偏移地址	名称	优先级	使能	说明
(0)	0x0000	—	—	—	栈顶初始地址
(1)	0x0004	Reset	–3	1	复位（优先级最高）
-14(2)	0x0008	NMI	–2	1	不可屏蔽中断
-13(3)	0x000C	HardFault	–1	1	硬件故障
-12(4)	0x0010	MPUFault	可编程	可编程	MPU 故障
-11(5)	0x0014	BusFault	可编程	可编程	总线故障
-10(6)	0x0018	UsageFault	可编程	可编程	应用故障
(7～10)	0x001C	—	—		保留
-5(11)	0x002C	SVCall	可编程	1	系统服务调用
-4(12)	0x0030	DebugMonitor	可编程	可编程	调试监控
-3(13)	0x0034	—	—		保留
-2(14)	0x0038	PendSV	可编程	1	挂起系统服务
-1(15)	0x003C	SysTick	可编程	1	SysTick 中断
0(16)	0x0040	GPIOA0	可编程	可编程	GPIOA0 中断
1(17)	0x0044	GPIOA1	可编程	可编程	GPIOA1 中断
2(18)	0x0048	GPIOA2	可编程	可编程	GPIOA2 中断
3(19)	0x004C	GPIOA3	可编程	可编程	GPIOA3 中断
4(20)	0x0050	—	—		保留
5(21)	0x0054	UARTA0	可编程	可编程	UARTA0 中断
6(22)	0x0058	UARTA1	可编程	可编程	UARTA1 中断
7(23)	0x005C	—	—		保留
8(24)	0x0060	I2CA0	可编程	可编程	I2CA0 中断
9～13	0x0064	—	—		保留
14(30)	0x0078	ADC_CH0	可编程	可编程	ADC_CH0 中断
15(31)	0x007C	ADC_CH1	可编程	可编程	ADC_CH1 中断
16(32)	0x0080	ADC_CH2	可编程	可编程	ADC_CH2 中断
17(33)	0x0084	ADC_CH3	可编程	可编程	ADC_CH3 中断
18(34)	0x0088	WDT	可编程	可编程	WDT 中断
19(35)	0x008C	GPTA0A	可编程	可编程	GPTA0A 中断
20(36)	0x0090	GPTA0B	可编程	可编程	GPTA0B 中断
21(37)	0x0094	GPTA1A	可编程	可编程	GPTA1A 中断
22(38)	0x0098	GPTA1B	可编程	可编程	GPTA1B 中断
23(39)	0x009C	GPTA2A	可编程	可编程	GPTA2A 中断
24(40)	0x00A0	GPTA2B	可编程	可编程	GPTA2B 中断

续表

中断号（矢量号）	偏移地址	名称	优先级	使能	说明
25～28	0x00A4	—	—		保留
29(45)	0x00B4	Flash	可编程	可编程	FLASH 中断
30～34	0x00B8	—	—		保留
35(51)	0x00CC	GPTA3A	可编程	可编程	GPTA3A 中断
36(52)	0x00D0	GPTA3B	可编程	可编程	GPTA3B 中断
37～45	0x00D4	—	—		保留
46(62)	0x00F8	uDMASoftware	可编程	可编程	DMA 软件中断
47(63)	0x00FC	uDMAError	可编程	可编程	DMA 错误中断
48～147	0x0100	—	—		保留
148(164)	0x0290	SHA	可编程	可编程	SHA 中断
149～150	0x0294	—	—		保留
151(167)	0x029C	AES	可编程	可编程	AES 中断
152(168)	0x02A0	—	—		保留
153(169)	0x02A4	DES	可编程	可编程	DES 中断
154～158	0x02A8	—	—		保留
159(175)	0x02BC	SDHOST	可编程	可编程	SDHOST 中断
160(176)	0x02C0	—	—		保留
161(177)	0x02C4	I2S	可编程	可编程	I2S 中断
162(178)	0x02C8	—	—		保留
163(179)	0x02CC	CAMERA	可编程	可编程	CAMERA 中断
164～170	0x02D0	—	—		保留
171(187)	0x02EC	NWP	可编程	可编程	NWP 中断
172(188)	0x02F0	PRCM	可编程	可编程	PRCM 中断
173～174	0x02F4	—	—		保留
175(191)	0x02FC	SSPI	可编程	可编程	SSPI 中断
176(192)	0x0300	GSPI	可编程	可编程	GSPI 中断
177(193)	0x0304	LSPI	可编程	可编程	LSPI 中断
178～239	0x0308	—	—		保留

附录 G

CC3200 μDMA 通道分配

通道号	通道编码			
	0	1	2	3
0	GPTA0_A	SHAMD5_CIN	—	SW（软件）
1	GPTA0_B	SHAMD5_DIN	—	SW（软件）
2	GPTA1_A	SHAMD5_COUT	—	SW（软件）
3	GPTA1_B	DES_CIN	—	SW（软件）
4	GPTA2_A	DES_DIN	I2S_RX	SW（软件）
5	GPTA2_B	DES_DOUT	I2S_TX	SW（软件）
6	GPTA3_A	GSPI_RX	GPIOA2	SW（软件）
7	GPTA3_B	GSPI_TX	GPIOA3	SW（软件）
8	UARTA0_RX	GPTA0_A	GPTA2_A	SW（软件）
9	UARTA0_TX	GPTA0_B	GPTA2_B	SW（软件）
10	UARTA1_RX	GPTA1_A	GPTA3_A	SW（软件）
11	UARTA1_TX	GPTA1_B	GPTA3_B	SW（软件）
12	LSPI_RX	—	—	SW（软件）
13	LSPI_TX	—	—	SW（软件）
14	ADC_CH0	—	SDHOST_RX	SW（软件）
15	ADC_CH1	—	SDHOST_TX	SW（软件）
16	ADC_CH2	GPTA2_A	—	SW（软件）
17	ADC_CH3	GPTA2_B	—	SW（软件）
18	GPIOA0	AES_CIN	I2S_RX	SW（软件）
19	GPIOA1	AES_COUT	I2S_TX	SW（软件）
20	GPIOA2	AES_DIN	—	SW（软件）
21	GPIOA3	AES_DOUT	—	SW（软件）
22	CAMERA	—	—	SW（软件）
23	SDHOST_RX	GPTA3_A	GPTA2_A	SW（软件）
24	SDHOST_TX	GPTA3_B	GPTA2_B	SW（软件）
25	SSPI_RX	I2CA0_RX	—	SW（软件）
26	SSPI_TX	I2CA0_TX	—	SW（软件）
27	—	GPIOA0	—	SW（软件）
28	—	GPIOA1	—	SW（软件）
29	—	—	—	SW（软件）
30	GSPI_RX	SDHOST_RX	I2CA0_RX	SW（软件）
31	GSPI_TX	SDHOST_TX	I2CA0_TX	SW（软件）

附录 H

实 验 指 导

实验 1　GPIO 实验

一、实验目的

1．掌握 GPIO 的使用方法。

2．熟悉 CCS 的使用，特别是程序的调试方法。

二、实验内容

1．通过 GPIO 控制 LED 的显示。

2．通过 GPIO 读取按键的状态。

三、实验程序

1．通过 GPIO 控制 LED 的显示程序参考 Blinky 项目。

2．通过 GPIO 读取按键的状态程序参考 3.1.3 节按键控制 LED 闪烁时间程序设计。

四、思考问题

1．GPIO 的基本操作有哪些？

2．调试的目的是什么？方法有哪些？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 2　UART 实验

一、实验目的

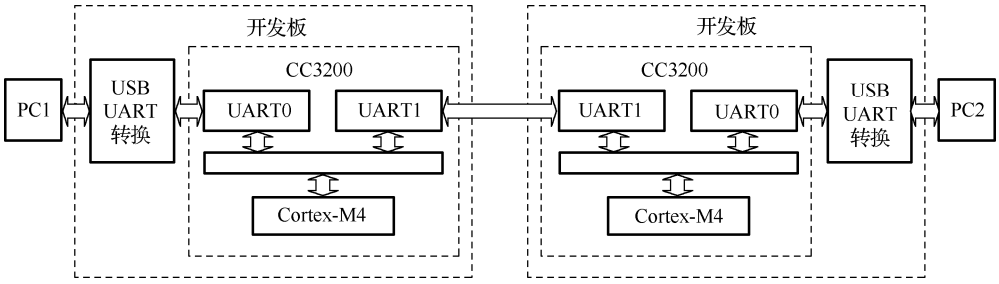
1．掌握 UART 的使用方法。

2．掌握 CCS 的的使用，特别是程序的调试方法。

二、实验内容

1．通过 UART 实现 CC3200 和 PC 间的字符收发。

2．通过 UART 实现两个 CC3200 间的字符收发，原理框图如下图所示。



提示：UART1_TX 和 UART1_RX 分别使用 PIN_58/59 的 PIN_MODE_6。

三、实验程序

1．通过 UART 实现 CC3200 和 PC 间的字符收发程序，参考 Uart_demo 项目。

2．通过 UART 实现两个 CC3200 间的字符收发程序，参考 Uart_demo 项目自行设计。

四、思考问题

1．UART 的主要指标有哪两个？

2．UART 的参数如何配置？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 3　SPI 实验

一、实验目的

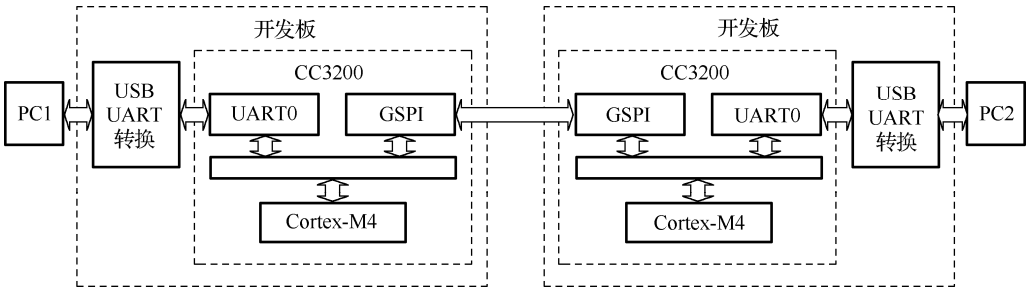
1．掌握 SPI 的使用方法。

2．比较 SPI 和 UART 使用方法的相同和不同。

二、实验内容

1．通过 SPI 实现自环回数据收发。

2．通过 SPI 实现两个 CC3200 间的数据收发，原理框图如下图所示。



三、实验程序

1．通过 SPI 实现自环回数据收发程序，参考 3.3.1 节 SPI 示例程序分析。

2．通过 SPI 实现两个 CC3200 间的数据收发程序，参考 Spi_demo 项目。

四、思考问题

1．SPI 数据收发和非阻塞数据收发的主要区别是什么？

2．SPI 和 UART 的使用方法有什么相同和不同？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。



实验 4　I2C 实验

一、实验目的

1．掌握 I2C 的使用方法。

2．掌握通过 I2C 实现对 I2C 器件操作的方法。

二、实验内容

1．I2C 的 4 种操作方法实现。

2．通过 I2C 实现温度测量和加速度测量。

三、实验程序

1．I2C 4 种操作方法的实现程序，参考 I2c_demo 项目。

2．通过 I2C 实现温度测量和加速度测量程序，参考 3.4.3 节温度测量和加速度测量程序设计。

四、思考问题

1．I2C 数据读写格式有什么不同？

2．I2C 的使用方法与 UART 和 SPI 有什么不同？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。



实验 5　GPT 实验

一、实验目的

1．掌握 GPT 基本功能的使用方法。

2．掌握 GPT PWM 和捕捉功能的使用方法。

二、实验内容

1．GPT 基本功能的实现。

2．GPT PWM 功能的实现。

3．GPT 捕捉功能的实现。

4．矩形波周期和脉宽测量。

三、实验程序

1．GPT 基本功能的实现程序，参考 Timer 项目。

2．GPT PWM 功能的实现程序，参考 PWM 项目。

3．GPT 捕捉功能的实现程序，参考 Timer_cc 项目。

4．矩形波周期和脉宽测量程序，参考 4.1.5 节矩形波周期和脉宽测量程序设计。

四、思考问题

1．GPT 的基本操作有哪些？

2．GPT PWM 和捕捉的操作主要有哪些？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 6　ADC 实验

一、实验目的

1．掌握 ADC 的使用方法。

2．掌握模拟电压的测量方法。

二、实验内容

1．ADC 基本功能的实现。

2．模拟电压测量。

提示：模拟电压通过电阻分压获得，测量电压分别用 ADC 采样值和电压值显示。

方法：将一个固定电阻和一个光敏电阻串联，固定电阻的一端接 3.3 V 电源，光敏电阻的一端接地，固定电阻和光敏电阻的连接处接 CC3200 的模拟输入引脚 PIN_57～60。固定电

阻阻值的选择要使没有光照射光敏电阻时模拟输入的电压值小于 1.4 V。改变照射光敏电阻的光强度，即可改变模拟输入引脚 PIN_57～60 的电压值。

注意：CC3200 ADC 的满量程电压是 1.4 V。

三、实验程序

1．ADC 基本功能的实现程序，参考 ADC 项目。

2．模拟电压测量程序，参考 ADC 项目自行设计。

四、思考问题

1．ADC 的基本操作有哪些？

2．如何将 ADC 采样值转换为电压值？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 7　NVIC 实验

一、实验目的

1．理解中断优先级分组和中断优先级的使用方法。

2．掌握外设中断的使用方法。

二、实验内容

1．中断优先级分组和中断优先级的实现。

2．GPIO 中断的实现。

三、实验程序

1．中断优先级分组和中断优先级的实现程序，参考 Interrupt 项目。

2．在 3.1.3 节按键控制 LED 闪烁时间程序设计的基础上，参考 Spi_demo 项目和 Timer 项目中的中断实现方法自行设计 GPIO 中断的实现程序。

四、思考问题

1．中断处理和子程序调用的本质区别有哪些？

2．外设中断的操作有哪些？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 8　μDMA 实验

一、实验目的

1．掌握外设 DMA 操作的使用方法。

2．理解μDMA 传输不同模式的特点。

二、实验内容

1．UART DMA 操作的实现。

2．μDMA 传输不同模式的实现。

三、实验程序

1．UART DMA 操作的实现程序，参考 Uart_dma 项目。

2．μDMA 传输不同模式的实现程序，参考 uDMA 项目。

四、思考问题

1．μDMA 传输的主要参数有哪些？

2．μDMA 传输不同模式的特点是什么？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 9　WLAN 站点实验

一、实验目的

1．掌握 CC3200 站点连接接入点的基本方法。

2．理解 WLAN 连接事件和获得 IPv4 事件的处理过程。

二、实验内容

1．CC3200 站点连接接入点。

2．检查 WLAN 连接和 Internet 连接。

三、实验程序

实验程序参考 Wlan_station 项目。

四、思考问题

1．CC3200 站点连接接入点最主要的操作是什么？

2．WLAN 连接和获得 IPv4 事件数据分别有哪些？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 10　WLAN 接入点实验

一、实验目的

1．掌握 CC3200 接入点设置和等待 Wi-Fi 站点连接的基本方法。

2．理解 WLAN 站点已连接事件和分配 IP 事件的处理过程。

二、实验内容

1．设置 CC3200 为接入点模式并等待 Wi-Fi 站点连接。

2．检查 Wi-Fi 站点连接。

三、实验程序

实验程序参考 Wlan_ap 项目。

四、思考问题

1．CC3200 接入点设置和等待 Wi-Fi 站点连接最主要的操作是什么？

2．分配 IP 事件数据有哪些？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 11　TCP 实验

一、实验目的

1．掌握 TCP 服务器和客户端的连接方法。

2．掌握 TCP 数据交换的方法。

二、实验内容

1．建立 TCP 服务器和客户端连接。

2．进行 TCP 数据交换。

三、实验程序

实验程序参考 Tcp_socket 项目。

四、思考问题

1．TCP 服务器和客户端的连接操作有哪些？

2．为什么接受 TCP 连接和连接 TCP 服务器必须使用阻塞模式，而接收 TCP 数据包通常使用非阻塞模式？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。



实验 12　UDP 实验

一、实验目的

1．掌握 UDP 数据交换的方法。

2．理解 UDP 和 TCP 的特点。

二、实验内容

1．创建 UDP 套接字。

2．进行 UDP 数据交换。

三、实验程序

实验程序参考 Udp_socket 项目。

四、思考问题

1．UDP 数据交换的操作有哪些？

2．UDP 和 TCP 各有什么特点？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。



实验 13　无线 UART 实验

一、实验目的

1．掌握无线 UART 的工作过程和设计方法。

2．掌握利用 TCP 收发数据的方法。

二、实验内容

利用两个 CC3200 间的 Wi-Fi 连接实现两个数据终端间的 UART 传输。

三、实验程序

实验程序参考 6.3 节无线 UART 程序设计。

四、思考问题

1．TCP 套接字连接中服务器和客户端可以对调吗？

2．与实验 2 中实验内容 2 比较，无线 UART 的特点是什么？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 14　AP 配置实验

一、实验目的

1．理解 HTTP 服务器的工作原理。

2．掌握 HTTP 服务器回调函数的实现方法。

3．理解网页中用户定义标记的实现方法

二、实验内容

用 AP 模式配置接入点参数。

三、实验程序

实验程序参考 provisioning_ap 项目。

四、思考问题

1．HTTP 服务器事件数据和响应数据主要有哪些？

2．网页中用户定义标记值如何获取和处理？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

实验 15　综 合 实 验

一、实验目的

1．掌握通过网页控制 CC3200 外设输出数据的方法。

2．掌握通过网页控制 CC3200 外设输入数据的方法。

3．掌握 HTTP 服务器的工作原理。

二、实验内容

1．实现通过网页控制 LED。

2．实现在网页中显示传感器的状态。

3．实现网页控制和状态显示的结合。

4．实现状态显示和网页控制的结合。

三、实验程序

1．通过网页控制 LED 的实现方法，参考 Out_of_box 项目的家庭自动化部分。

2．在网页中显示传感器状态的实现方法，参考 Out_of_box 项目的家用电器部分。

3．网页控制和状态显示结合的实现方法，参考 Out_of_box 项目的安全保障部分。

4．状态显示和网页控制结合的实现方法，参考 Out_of_box 项目的智能能源部分。

四、思考问题

1．通过网页控制 CC3200 外设输出数据的步骤有哪些？

2．通过网页网页 CC3200 外设输入数据的步骤有哪些？

3．GET 请求和 POST 请求的处理方法有哪些？各有什么特点？

五、实验报告

1．实验目的。

2．实验内容。

3．硬件方框图和电路图。

4．软件流程图和核心语句。

5．设计过程中遇到的问题和解决方法。

6．思考问题解答、收获和建议等。

使 用 软 件

[1] CCS v6.1.0：CCS6.1.0.00104_win32.zip

[2] CC3200 SDK v1.1.0：CC3200SDK-1.1.0-windows-installer.exe

[3] CC31xx_CC32xx_ServicePack-1.0.0.10.0-windows-installer.exe

[4] PinMux v3.0：pinmux-3.0.334-setup.exe

[5] UniFlash v3.2：uniflash_cc3xxx_setup_3.2.0.00123.exe

[6] Tera Term v4.85：teraterm485.exe

[7] Iperf v1.11：Iperf.apk

[8] SmartConfig：SmartConfigCC3X.apk

参 考 文 献

[1] Texas Instruments Incorporated, CC3200 SimpleLink™ Wi-Fi® and IoT Solution,a Single-Chip Wireless MCU(SWAS032E), 2014.6

[2] Texas Instruments Incorporated, CC3200 SimpleLink™ Wi-Fi® and IoT Solution,a Single-Chip Wireless MCU Programmer's Guide(SWRU369), 2014.6

[3] Texas Instruments Incorporated, CC3200 SimpleLink™ Wi-Fi® and IoT Solution With MCU LaunchPad Hardware User's Guide(SWRU372), 2014.6

[4] Texas Instruments Incorporated, CC3200 SimpleLink™ Wi-Fi® and IoT Solution With MCU LaunchPad Getting Started Guide(SWRU376A), 2014.6

[5] Texas Instruments Incorporated, CC3200 SimpleLink™ Wi-Fi® and IoT Solution,a Single-Chip Wireless MCU Technical Reference Manual(SWRU367), 2014.6

[6] Texas Instruments Incorporated, CC3100/CC3200 SimpleLink™ Wi-Fi® Network Processor Subsystem Programmer's Guide(SWRU368), 2014.6

[7] Texas Instruments Incorporated, CC3200 Peripheral Driver Library User's Guide, 2014.8

[8] Texas Instruments Incorporated, CC32XX SimpleLink Host Driver Library Programmer's Guide, 2014.8

[9] 郭书军，王玉花．ARM Cortex-M3 系统设计与实现．北京：电子工业出版社，2014．

[10] [美] David Gourley，Brian Totty，Marjorie Sayer，等著．HTTP 权威指南．陈涓，赵振平，译．北京：人民邮电出版社，2012．